WWWC Topics:

Perl/CGI Tutorial

Overview

This tutorial provides a problem-oriented introduction to Perl within the context of CGI. The approach is narrative. A problem is introduced and the tutorial then proceeds, step by step, to solve the problem. Perl concepts and features are introduced as they are needed to accomplish each step. The Perl Basics discussion, by contrast, provides a more thorough description in which concepts and features are presented in a logical order consistent with the structure of the Perl language, itself. The two discussions are intended to complement one another.

The problem that will be solved is extracting the data passed to a program by a WWW server through the Common Gateway Interface (CGI) and constructing a reply expressed in HTML that is passed by the program back through the interface to the server and, then, to the client/user for display.

The discussion assumes familiarity with UNIX but no prior experience with Perl.

Steps

I have broken the problem down into the following steps:

1. Perl framework and mechanics
2. Hello, World
3. Hello, World, from CGI
4. Hello, World, from CGI, in HTML
5. Echo Environment Variables
6. Echo STDIN Variables
7. Perl Library

---

1. Perl framework and mechanics

For your Perl work in this course, you will need to setup one or two directories. Create a directory within the `wwwc-bin` directory and give it your last name as its name, like your directory in members. Put your CGI scripts there, once you have debugged the basic Perl code. While you are working on the Perl code, you may wish to use a different directory for those versions of your programs and then move each one into your directory under `wwwc-bin` for CGI and HTML testing and actual use.

Each time you create a new Perl program, you must make it executable. You do this using the `chmod +x` *`filename`* command.

Your Perl program begins with an invocation of the Perl interpreter in the first line. By convention, the line begins with a hash or pound character (#), followed by an exclamation point (!), followed by the path to the interpreter.

Comments may appear anywhere on a line following a hash (#) symbol.

Perl statements end with a semicolon (;), and whitespace (spaces, tabs, etc.) may be used freely.

Perl programs do not require an end mark or end statement (other than the end of file for the program, itself).

The following is a null Perl program that is valid within the UNC Department of Computer Science UNIX environment:

```
#!/usr/local/bin/perl
# a comment
; # a null statement
```

I suggest you type it into a file, make it executable, and run it.

---

2. Hello, World

The program in the preceding step, as you saw, didn't do anything, unless you mistyped something and got an error message. The goal here is to get Perl to give a minimal response so that we know it is alive. We'll do this with a `print` statement.

The print statement begins with the keyword, `print`, followed by what is to be printed, followed by the required semicolon.

What you print will usually be placed within double quote marks (""; yes, Perl makes a distinction between single quotes ('') and double quotes (""), which will be explained in the Perl Basics discussion). It may also be placed within parentheses, but they are not required.

If you would like for the output to be placed on a line by itself, print `\n` at the end of the line.

Hello, World program

```
#!/usr/local/bin/perl
print "Hello, World!\n";
```

Type it in and run it.

---

3. Hello, World, from CGI

The next step is to modify your `Hello, World` program so that it includes the required CGI header lines and then copy it into a directory where it can be run from within the Web, as opposed to UNIX and Telnet.

Recall two things from the discussion of CGI. First, output is sent from the program to the server via `STDOUT`; consequently, normal Perl print statements can be used to send the data. Second, the data from the CGI program must begin with several header lines followed by a blank line. Those header lines include that `Status` and the *Content-type*

The Status line includes two fields: a numeric return code and an explanation. We'll return values of "200" and "ok", on a line by themselves.

The Content-type describes the type of data that will be sent, expressed as a MIME type/subtype form. Since our data is text and will not be formatted in HTML, we'll use "text/plain".

Finally, we'll put out a blank line to separate the header lines from the data produced by the program.

Hello, World, from CGI program

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

print "200 OK\n";
print "Content-type: text/plain\n";
print "\n";

print "Hello, World, from CGI\n";
```

I suggest you begin with your Hello, World program and modify it. Then test it as a conventional Perl program. After you see that it is creating the output you want, then copy it into the CGI directory.

To do so, copy your `Hello, World, from CGI` program into the directory under `wwwc-bin` you created for your CGI scripts and to which you gave your last name as its name.

Run the program. To do this, provide a Web browser with a URL that is the path to your program. In our current working context, the URL will look like this:

http://www2.cs.unc.edu/wwwc-bin/*last_name/filename*

where *last_name* is your directory for your CGI programs, and *filename* is the name of your Perl `Hello, World, from CGI` program.

If yours doesn't work for some reason, you can execute mine (Hello, World, from CGI).

---

4. Hello, World, in HTML

In the previous step, we generated CGI header lines and a single plain text "content" line. In this step, we'll expand the content portion and embed HTML tags within it. As a result, when the data arrive at the browser, they can be formatted and displayed as conventional HTML data. Again, we'll keep the changes to a minimal, adding just the tags required for a proper HTML document to our old refrain.

You may wish to begin with your `Hello, World, from CGI` script and edit it. If you do, you should change the `content-type` from `text/plain` to `text/html` since your program will be generating HTML.

There's really nothing to generating the HTML data. You "write" it just like you normally do for a conventional, static document, except that each line is placed inside a `print` statement and contains an explicit newline (`\n`) character, if that is desired.

Following is a `Hello, World, in HTML` program, set up like one of the standard pages used for course materials.

Hello, World, in HTML program

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

print "200 ok\n";
print "content-type: text/html\n\n";

print "<HTML>\n";

print "<HEAD>\n";
print "<TITLE>hello, world html </TITLE>\n";
print "<H1>Hello, World, in HTML </H1>\n";
print "</HEAD>\n";

print "<BODY>\n";
print "<HR>\n";
print "Hello, World, in HTML\n";
print "</BODY>\n";
```

Write your own `Hello, World, in HTML` script, test it, and put it in your `wwwc-bin` directory. If you like, you can execute mine (`Hello, World, in HTML`).

---

5. Echo Environment Variables

In steps 5 & 6, we will go a little further into the programming facilities provided by Perl. The discussion will be limited, primarily, to those needed to accomplish the task. For context and related features, see the Perl Basics discussion or a standard text, such as *Learning Perl.*

Recall from the discussion of CGI that the server places values in set of `environment variables` which can be accessed from within your CGI program as a special kind of global variable. For programs invoked with the method, GET, they provide the only means of passing data to the program. For programs that use POST, data from the user are passed through STDIN, which will be discussed next. However, HTTP protocol data are passed through environment variables for both GET and POST methods.

The server makes environment variables available to CGI programs in different ways according to the programming language in which the programs are written. The discussion here is concerned only with handling environment variables with Perl.

The primary goal for step 5 is understanding what the environment variable data looks like and how it can be accessed by a Perl program. We won't do anything with the data except print it as a formatted list. You may wish to refer to the `Echo Environment Variables` program, below, during the discussion. Note that most of the program is similar to the Hello, World, in HTML program with respect to HTML boilerplate. The lines to focus on are near the bottom where data for an `unorderd list` are generated. Within the beginning and end tags are two Perl statements. Those two statements are our concern here.

There is a good deal of magic expressed in those two statements. They will make a lot more sense, as will the code in step 6 that follows, if we pause for a moment and talk about Perl variables and names. The initial character of a Perl name identifies the particular type of variable or entity:

$name
: *scalar* variable, either a number or string; Perl does not differentiate between the two, nor does it differentiate between integers and reals.

@name()
: *array* ; Perl uses parentheses with arrays, but other delimiters with other kinds of variables, as discussed below. However, while Perl uses the "at" symbol and parentheses with respect to the array as a whole, individual elements within an array are referred to as scalars, and the index is placed in square brackets (i.e., $name[0] is the first element of @name).

%name{}
: *associative array* ; a special, 2-dimensional array, ideal for handling attribute/value pairs. The first element in each row is a `key` and the second element is an associated value. Instead of using a number to index the array, you use a key value, such as $name{"QUERY_STRING"}, to reference the value associated with that particular key, in this case QUERY_STRING. Since the associated value is a scalar, the variable has a $ prefix. Note, also, the use of curly braces (`{}`) as delimiters.

&name()
: *function* ; the ampersand is placed before the name when the function is called; if the function takes arguments, they are placed within parentheses following the name of the function.
When the functions is defined, the name is preceded by the key word, `sub`, but does not have the ampersand prefix.; code for the subroutine is then placed within curly braces ({}) following the name.

We can now go back to the program. Look at the line two-thirds of the way down that begins with the keyword, `foreach`. We'll unwind it first.

The server provides the environment variables to the CGI Perl program in the form of a special *associative array* it creates, called `%ENV`. Each row of `%ENV`, then, contains a key, which is the name of an attribute, and the value that is associated with that attribute.

`keys` is a built-in function that takes an associative array and returns a list (one-dimensional array) of its keys. Thus, one would expect to see the expression, `key (%ENV)`; Perl makes the parentheses optional with `key` and they are omitted here.

`$key` is a scalar variable that can receive a specific key value, which it does by virtue of the `foreach` operator that precedes it in the line. `foreach`takes a list of values and assigns them, one by one, to the scalar variable that follows it.

Thus, to paraphrase the whole line: `$key` iterates through the list of keys produced by the built-in function, `key`, from the associative array, `%ENV`, built by the server.

On to the next line, in which keys and associated values are printed. The line is relatively straight-forward once one thing is understood: Perl interprets variables within double quotation marks. Consequently, the print statement begins by printing the HTML tag for a new list item. It then interprets `$key` according to its current key value, which was assigned, iteratively, in the preceding `foreach` statement. It next prints the equal sign. Finally, it prints the array value indexed by the current `$key` value. Note that this array value is referred to as $ENV{$key}. The dollar sign prefix is used since only a single, or scalar, value is being referenced. Note, also, the use of curly braces, since the whole thing is an associative array.

That's a lot to swallow, perhaps, in two lines of code, but such is the nature of Perl -- *very succinct*, but also very powerful. There's an initial hump to get over, but not all that high. Then you can begin the long climb toward more and more sophisticated uses of Perl, if it's to your taste.

Echo Environment Variables Program

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

print "200 ok\n";
print "content-type: text/html\n\n";

print "<HTML>\n";

print "<HEAD>\n";
print "<TITLE>echo cgi env. vars.</TITLE>\n";
print "<H1>Echo CGI Environment Variables</H1>\n";
print "</HEAD>\n";

print "<BODY>\n";
print "<HR>\n";
print "<H3>Environment Variables</H3>\n";
print "<UL>\n";
foreach $key (keys %ENV) {
  print "<LI>$key = $ENV{$key}\n";
  }
print "</UL>\n";
print "</BODY>\n";

print "</HTML>\n";
```

Write and test an `Echo Environment Variables` script. You can also execute mine (`Echo Environment Variables`).

6. Echo STDIN Variables

The task in step 6 is to *parse* the data passed from the server to a CGI program through STDIN as a character string. Passing data through STDIN is used by METHOD=POST, as opposed to passing them through the environment variable, QUERY_STRING, as is the case when METHOD=GET. Parsing is required because the data passed between client and server is compressed into a continuous string, without spaces, and some characters have been mapped to other values.

The purpose of parsing is to break the string into attribute/value pairs, translate various special characters that were coded for transit back into their original character forms, and to store the translated attribute/value pairs in a convenient data structure, i.e., an associative array. Although the point was skipped in step 5, parsing is also needed there for the character string passed through the QUERY_STRING environment variable when METHOD=GET.

Parsing strings in CGI, for data passed through STDIN or through QUERY_STRING, requires four steps:

A. TOKENIZE the string into a list of attribute/value strings
B. SPLIT each attribute/value token into separate key and value
C. DEPLUS the strings to translate plus signs(+) into spaces
D. DECODE 3-character hex representations of special characters and translate them back into their original 1-character forms

The order in which these four steps are carried out matters. For example, decode should be done last. Ampersands are used to separate attribute/value strings from one another; consequently, ampersands in the data are translated into 3-character hex values. If you translated these hex values back to ampersands before tokenizing, they would be confused with the ampersands used as delimiters. The order generally recommended is that given above. One exception is to deplus first, while the input data exist as a single string, before tokenizing and splitting, which is the strategy shown below.

Parsing, as performed in the program below, requires some five or six new Perl concepts and operators:

- <STDIN>
- Translate and squeeze
- Split
- Associative array assignment
- Substitution

Each will be discussed in the context of the parse program, below.

`<STDIN>` is actually an operator. It returns the next line of input from the file, STDIN. Consequently, it does not need to be used with another operator or verb, such as a `read`. Consequently, the command

```
$in_string = <STDIN>;
```

reads the next line of input, which is the entire concatenated string of attributes and values, and places them in the scalar variable, `$in_string`.

`Translate and squeeze`. The next section of code translates plus signs (+), used to indicate spaces in the original data, back into spaces; it also removes multiple spaces so that only a single space exists between any two words. The Perl command used for this is `tr`, for translate. It takes two patterns, delimited by slash (/) characters, and translates instances of the first into the second. For example, it can be used to translate all uppercase characters to lowercase, or vice versa. Patterns can be quite complex, and will be discussed in more detail when *regular expressions* are described.

In the line of code shown, the plus sign is preceded by a backslash (\) to indicate that it is the character, plus, in this context and not the Perl operator for addition. The `s` at the end of the expression removes, or squeezes out, multiple instances of the translated pattern, spaces in this case. Finally, the symbol `=~` is actually an operator. It identifies the variable on the left as the one to which the operator on the right, the `tr`, is applied. Thus, it works like an assignment statement, although it is not literally that. Had it not been used, the translate would have been applied to an invisible (predefined) variable, called the default variable and denoted `$_`. It is a somewhat mysterious variable whose value is set as a result of the last operation; often it is the variable to which one would apply the next operation.

`Split` does what the name implies: it takes a pattern, shown between the slash (/) delimiters, and a character string, and returns a list of the portions of the string that precede and succeed the pattern. Thus, it produces a list of the portions of the string that *don't* match the pattern and throws away the portions that do match.

In the TOKENIZE step, the input string is split on the pattern, /&/ and the resulting list of attribute/value strings is assigned to an array, `@attr_val_strings`, indicated by the at-sign (@) prefix of the variable name.

In the SPLIT step that follows (an unfortunate choice of labels on my part), each such string is further split into the portions that come before and after an equal (=) sign, with the two strings assigned to a 1x2 array, `@pair`. Element `$pair[0]` is the part that comes before the equal sign, and `$pair[1]` is the part that comes after. In the next line of code, these two array values are assigned as the associated key and value parts for a row in the associative array, `%attr_value`. However, since the expressions refers to individual elements of the array, each such element is referred to using its scalar prefix. Let me paraphrase the line,

```
$attr_value{$pair[0]} = $pair[1];
```

Assign the contents of $pair[1], the part of the string that came after the equal sign, as the value element in the row of the associative array, `%attr_value`, that is indexed by the key, $pair[0], which is the part of the string that came before the equal sign. Since the assignment applies to only a single element in the array, the scalar name, $attr_value, is used.

`Associative array assignment`. Just did it.

`Substitution`. The substitution operator, `s`, is at the center of the DECODE step. This is the Matterhorn. Once we get over this peak, it's all downhill form there. As with many Perl expressions, there is a great deal of magic packed into this single line of code. That's the beauty of the language, if you like it, or its downfall, if you don't. But it is one of the main characteristics that makes Perl what it is. We'll work from the outside in.

The DECODE block of code looks for 3-character sequences that consist of an escape character, `%`, followed by a 2-character hexadecimal value. Special characters, such as parentheses, spaces, ampersands, and the like, that might interfere with processing the data string, are coded in this way for transfer; they must be translated back to their original forms for processing. That is what DECODE does.

The code to do this begins innocently enough, using the `key` operator to return the list of keys from the associative array, `%attr_value` and the `foreach` operator to step through that list, referencing each key value in turn through the scalar, `$key`. In the next two statements, the substitution magic is done to translate all of the 3-character hex codes back into their original 1-character forms. The first substitution is performed on the `$key` and, hence, on the key element of the associative array, `%attr_value`. The second uses the transformed key as the index into the associative array and transforms the corresponding value element. Thus, each key and each corresponding value are transformed separately, requiring two substitution operations for the pair.

Now for the assault on the peak. The substitution operator, like the translate operator, takes two patterns, delimited by slashes. It looks for an instance of the first pattern in the target string and substitutes an instance of the second pattern for it. The pattern that is looked for here is `%(..)`. The percent sign is a literal and is looked for, explicitly. The two periods (`..`) are matched by any two characters. The parentheses around the two periods tells Perl to "remember" those two characters so that they can be referred to later, in this context through the variable, `$1`. Thus, the string, `%28`, which is the coded representation for a left parenthesis, would be matched by this pattern and the `28` would be assigned as the value to the variable, `$1`. When such a pattern is found, the operator substitutes what follows, delimited by the slashes, for the 3-character string.

What is substituted here is `pack("c",hex($1))`. `pack` takes two arguments, a format control string and a list of values, and creates a single string from those values. The format control string is defined to be a single character, denoted by the`"c"`, and the list of values is the single value, `$1`, which is the hex code for the character to be translated.

Note that what is produced as a result of the substitution is a Perl operator, `pack`. The final `e` tells Perl to *execute* that operation and substitute the results of the operation in the place where the pattern is found. The `g` at the end of the expressions says that the substitution should be made for all occurrences of the pattern. Finally, the `=~` operator directs the substitution to the desired string.

To sum up, the DECODE block goes through the associative array of attributes and their corresponding values one row at a time, looks for all instances of special characters -- coded as the escape character, `%`, followed by a 2-digit hex value -- and replaces each such 3-character sequence with the appropriate single (special) character; it does this, first, for each key in the associative array and, then, for the associated value indexed by that key.

Echo STDIN Variables Program

```
#!/usr/local/bin/perl
#
#     INPUT data
$in_string = <STDIN>
```

```
#
#     DEPLUS $in_string
$in_string =~ tr/\+/ /s; # translate and squeeze multiple spaces
#
#     TOKENIZE attr/val strings
@attr_val_strings = split (/&/, $in_string);
#
#     SPLIT attr/val strings and put into assoc. array
foreach $out_str (@attr_val_strings) {
@pair = split (/=/, $out_str);
$attr_value{$pair[0]} = $pair[1];
  }
#
#     DECODE special characters
foreach $key (keys %attr_value) {
  $key =~ s/%(..)/pack("c",hex($1))/ge;
  $attr_value{$key} =~ s/%(..)/pack("c",hex($1))/ge;
  }

#     OUTPUT section

#     generate header lines
print "200 ok\n";
print "content-type: text/html\n\n";

#     GENERATE report, in HTML
print "<HTML>\n";

print "<HEAD>\n";
print "<TITLE>stdin vars.</TITLE>\n";
print "<H1>Print CGI STDIN Variables</H1>\n";
print "</HEAD>\n";

print "<BODY>\n";
print "<HR>\n";
print "<H3>STDIN Variables</H3>\n";
print "<UL>\n";
foreach $key (keys %attr_value) {
  print "<LI>$key = $attr_value{$key}\n";
  }
print "</UL>\n";
print "</BODY>\n";

print "</HTML>\n";
```

Write and test an `Echo STDIN Variables` script. You can also execute mine (`Echo STDIN Variables`).

---

7. Perl Library

Now that we have climbed the peak once, we'll take the chair lift the next time. It is important that you understand both the details of how data is coded and passed to a CGI program and, in the context of this course, how to write Perl programs to process that data. However, parsing is a routine task as is creating HTTP headers, and we have not addressed issues such as error recognition and handling attributes that have multiple values. Other people have written Perl programs that perform routine tasks such as these. If you can find such programs, they can be placed in a local library and incorporated into your programs or executed directly. This allows you to build on their work, for routine tasks, and concentrate on the special processing your particular application requires.

I have placed one such set of programs, written by Steven E. Brenner and called `cgi-lib.pl`, in a library, called `lib`, under the course directory, `wwwc-f95`. You should begin this step by reading the Perl code for `cgi-lib.pl` to get a general sense of what is included. After that, refer to the program, below, to see how its functions can be used to parse CGI input, build appropriate data structures, and echo their values.

Echo Variables Using `lib` Program

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

require "/afs/unc/proj/wwwc-f95/lib/cgi-lib.pl";
#
print &PrintHeader;
#
print "<H2>Environment variables</H2>\n";
print &PrintVariables(%ENV);
print "<HR>\n";
#
print "<H2>User-defined variables</H2>\n";
#
 if (&ReadParse(*input)) {
    print &PrintVariables(%input);
 } else {
   print '<form><input type="submit">Data: <input name="myfield">';
}
```

If you would like to test the program, here are two forms to do so:

POST form
GET form

---

email: jbs@cs.unc.edu
url: http://www.cs.unc.edu/~jbs/