CMR
INSTITUTE OF
TECHNOLOGY

USN	1	C								

Internal Assessment Test 2 – May 2017

Sub:	Advanced Web Programming							Code:	13MCA43
Date:	09-05-17	Duration:	90 mins	Max Marks:	50	Sem:	IV	Branch:	MCA

Note: Answer any 5 questions. All questions carry equal marks. Total marks: 50

			Marks	OBE CO	OBE RBT
1	a.	Explain the different variables in Perl. Explain each with examples	[10]	CO3	L4

There are three types of variables in Perl. A **scalar** variable will precede by a dollar sign ($) and it can store either a number, a string, or a reference. An **array** variable will precede by sign @ and it will store ordered lists of scalars. Finally, the **Hash** variable will precede by sign % and will be used to store sets of key/value pairs.

Perl maintains every variable type in a separate namespace. So you can, without fear of conflict, use the same name for a scalar variable, an array, or a hash. This means that $foo and @foo are two different variables. Perl variables do not have to be explicitly declared to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

Scalar Variables

A scalar is a single unit of data. That data might be an integer number, floating point, a character, a string, a paragraph, or an entire web page. Simply saying it could be anything, but only a single thing.

```
#!/usr/bin/perl


$age = 25;          # An integer assignment
$name = "John Paul";   # A string
$salary = 1445.50;     # A floating point
```

Array Variables

An array is a variable that stores an ordered list of scalar values. Array variables are preceded by an "at" (@) sign. To refer to a single element of an array, you will use the dollar sign ($) with the variable name followed by the index of the element in square brackets.

```
#!/usr/bin/perl
@ages = (25, 30, 40);
@names = ("John Paul", "Lisa", "Kumar");
print "\$ages[0] = $ages[0]\n";
print "\$ages[1] = $ages[1]\n";
print "\$ages[2] = $ages[2]\n";
print "\$names[0] = $names[0]\n";
print "\$names[1] = $names[1]\n";
print "\$names[2] = $names[2]\n";
```

Hash Variables

A hash is a set of **key/value** pairs. Hash variables are preceded by a percent (%) sign. To refer to a single element of a hash, you will use the hash variable name followed by the "key" associated with the value in curly brackets.

Here is a simple example of using hash variables –

```
#!/usr/bin/perl
%data = ('John Paul', 45, 'Lisa', 30, 'Kumar', 40);
print "\$data{'John Paul'} = $data{'John Paul'}\n";
print "\$data{'Lisa'} = $data{'Lisa'}\n";
print "\$data{'Kumar'} = $data{'Kumar'}\n";
```

This will produce the following result –

```
$data{'John Paul'} = 45
$data{'Lisa'} = 30
$data{'Kumar'} = 40
```

2	a.	Explain the control structures in Perl with suitable examples	[10]	CO3	L4

While

```
 $a=0;

While($a<10) {
{
        Print $a;
         $a++;
}
```

Do-while

```
$a=0;
Do
{
Print $a;
$a++;
}
While($a<10);
```

For

```
For($a=0;$a<10;$a++)
{
   Print $a;
}
```

Foreach

```
@a = (“a”,”b”,”c”,10);
Foreach(@a as $k)
{
Print $k;
}
```

Until
 Unless
Next
Last
Nested loops
If
Elsif

3 a. **Pattern Matching / Regular Expressions** [10] CO3 L4

You can write about the types as well. POSIX and PERL style

A regular expression is a string of characters that defines the pattern or patterns you are viewing. The syntax of regular expressions in Perl is very similar to what you will find within other regular expression supporting programs.

The basic method for applying a regular expression is to use the pattern binding operators **=~** and **!~**. The first operator is a test and assignment operator.

There are three regular expression operators within Perl.

- Match Regular Expression - m//
- Substitute Regular Expression - s///
- Transliterate Regular Expression - tr///

The forward slashes in each case act as delimiters for the regular expression (regex) that you are specifying. If you are comfortable with any other delimiter, then you can use in place of forward slash.

The Match Operator

The match operator, m//, is used to match a string or statement to a regular expression.

```
#!/usr/bin/perl


$bar = "This is foo and again foo";
if ($bar =~ /foo/){
  print "First time is matching\n";
}else{
  print "First time is not matching\n";
}
```

The m// actually works in the same fashion as the q// operator series.you can use any combination of naturally matching characters to act as delimiters for the expression

Regular Expression Variables

Regular expression variables include **$**, which contains whatever the last grouping match matched; **$&**, which contains the entire matched string; **$`**, which contains everything before the matched string; and **$'**, which contains everything after the matched string. Following code demonstrates the result –

```
#!/usr/bin/perl
$string = "The food is in the salad bar";
$string =~ m/foo/;
print "Before: $`\n";
print "Matched: $&\n";
print "After: $'\n";
```

The Substitution Operator

The substitution operator, s///, is really just an extension of the match operator that allows you to replace the text matched with some new text. The basic form of the operator is –

```
s/PATTERN/REPLACEMENT/;
```

The PATTERN is the regular expression for the text that we are looking for. The REPLACEMENT is a specification for the text or regular expression that we want to use to replace the found text with. For example, we can replace all occurrences of **dog** with **cat** using the following regular expression –

```
#/user/bin/perl
$string = "The cat sat on the mat";
$string =~ s/cat/dog/;
print "$string\n";
```

When above program is executed, it produces the following result –

```
The dog sat on the mat
```

The Translation Operator

Translation is similar, but not identical, to the principles of substitution, but unlike substitution, translation (or transliteration) does not use regular expressions for its search on replacement values. The translation operators are –

```
tr/SEARCHLIST/REPLACEMENTLIST/cds
y/SEARCHLIST/REPLACEMENTLIST/cds
```

The translation replaces all occurrences of the characters in SEARCHLIST with the corresponding characters in REPLACEMENTLIST. For example, using the "The cat sat on the mat." string we have been using in this chapter –

```
#/usr/bin/perl
```

```
$string = 'The cat sat on the mat';
$string =~ tr/a/o/;
print "$string\n";
```

4	a.	**Subroutines**	[6]	CO3	L4

A Perl subroutine or function is a group of statements that together performs a task. You can divide up your code into separate subroutines, logically the division usually is so each function performs a specific task.

```
sub subroutine_name{
  body of the subroutine
}
```

The typical way of calling that Perl subroutine is as follows −

```
subroutine_name( list of arguments );
&subroutine_name( list of arguments );
```

```
#!/usr/bin/perl
# Function definition
sub Hello{
  print "Hello, World!\n";
}
# Function call
Hello();
```

Passing Arguments to a Subroutine

You can pass various arguments to a subroutine like you do in any other programming language and they can be acessed inside the function using the special array @_. Thus the first argument to the function is in $_[0], the second is in $_[1], and so on.

You can pass arrays and hashes as arguments like any scalar but passing more than one array or hash normally causes them to lose their separate identities. So we will use references (explained in the next chapter) to pass any array or hash.

```
#!/usr/bin/perl


# Function definition
sub Average{
  # get total number of arguments passed.
  $n = scalar(@_);
  $sum = 0;
  foreach $item (@_){
    $sum += $item;
  }
  $average = $sum / $n;
  print "Average for the given numbers : $average\n";
}
# Function call
&Average(10, 20, 30);
```

Passing Hashes to Subroutines

When you supply a hash to a subroutine or operator that accepts a list, then hash is automatically translated into a list of key/value pairs. For example −

```
#!/usr/bin/perl
# Function definition
sub PrintHash{
  my (%hash) = @_;

  foreach my $key ( keys %hash ){
    my $value = $hash{$key};
    print "$key : $value\n";
  }
}
%hash = ('name' => 'Tom', 'age' => 19);
# Function call with hash parameter
PrintHash(%hash);
```

Returning Value from a Subroutine

You can return a value from subroutine like you do in any other programming language. If you are not returning a value from a subroutine then whatever calculation is last performed in a subroutine is automatically also the return value.

You can return arrays and hashes from the subroutine like any scalar but returning more than one array or hash normally causes them to lose their separate identities.

```
#!/usr/bin/perl
# Function definition
sub Average{
 # get total number of arguments passed.
 $n = scalar(@_);
$sum = 0;
 foreach $item (@_)
{
    $sum += $item;
```

```
  }
  $average = $sum / $n;
  return $average;
}
# Function call
$num = &Average(10, 20, 30);
print "Average for the given numbers : $num\n";
```

b. Write a PERL script to populate an integer array and display all numbers greater than the average of the array — [4] CO3 L4

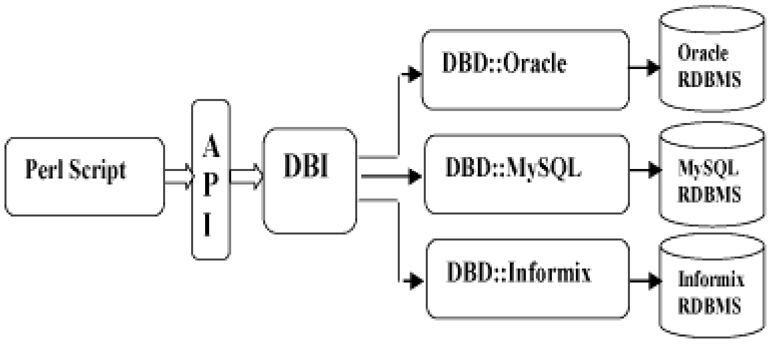
```
#!/usr/bin/perl
$n = <STDIN>;
For ($i=0;$i<$n;$i++)
{
     $item = <STDIN>;
    Push(@ar,$item);
}
$sum = 0;
For($i=0;$i<$n;$i++)
{
$sum += $ar[$i];
}

$avg = $sum/$n;
For($i=0;$i<$n;$i++)
{
If($ar[$i]>$avg)
{
    Print $ar[$i]."\n";
}
}
```

5 a. Explain the architecture of DBI. Discuss the database connection with a suitable example — [10] CO3 L4

DBI stands for **Database Independent Interface** for Perl which means DBI provides an abstraction layer between the Perl code and the underlying database, allowing you to switch database implementations really easily. The DBI is a database access module for the Perl programming language. It provides a set of methods, variables, and conventions that provide a consistent database interface, independent of the actual database being used.

Architecture of a DBI Application

DBI is independent of any database available in backend. You can use DBI whether you are working with Oracle, MySQL or Informix etc. This is clear from the following architure diagram.



```
my $driver = "mysql";
my $database = "TESTDB";
my $dsn = "DBI:$driver:database=$database";
my $userid = "root";
my $password = " ";
my $dbh = DBI->connect($dsn, $userid, $password )
```

If a connection is established with the datasource then a Database Handle is returned and saved into $dbh for further use otherwise $dbh is set to *undef* value

Insert

Prearing SQL statement with INSERT statement. This will be done using **prepare()** API.
Executing SQL query to select all the results from the database. This will be done using **execute()**API.
Releasing Stattement handle. This will be done using **finish()** API

READ Operation

READ Operation on any database means to fetch some useful information from the database ie one or more records from one or more tables. So once our database connection is established, we are ready to make a query into this database. Following is the procedure to query all the records having AGE greater than 20. This will take four steps

Preparing SQL SELECT query based on required conditions. This will be done using **prepare()**API.
Executing SQL query to select all the results from the database. This will be done using **execute()**API.
Fetching all the results one by one and printing those results.This will be done using **fetchrow()** API.
Releasing Stattement handle. This will be done using **finish()** API

Disconnecting Database
To disconnect Database connection, use **disconnect** API as follows:

```
$rc = $dbh->disconnect();
```

Similarly for update and delete operation

6	a.	Write a complete script for file uploading in PERL	[10]	CO3	L4

<u>Fileupload.html</u>

```
<HTML>
 <HEAD></HEAD>
 <BODY>
 <FORM ACTION="upload.cgi" METHOD="post" ENCTYPE="multipart/form-
data">
 Photo to Upload: <INPUT TYPE="file" NAME="photo">
 <br><br>
 <INPUT TYPE="submit" NAME="Submit" VALUE="Submit Form">
 </FORM>
 </BODY>
</HTML>
```

<u>Upload.cgi</u>

```
#!C:\Perl\bin\perl.exe
 use CGI;
 $upload_dir = "C:/xampp/htdocs/Trials/upload/";

$query = new CGI;
print $query->header( );
$filename = $query->param("photo");
$filename =~ s/.*[\/\\](.*)/$1/;
$upload_filehandle = $query->upload("photo");

open UPLOADFILE, ">$upload_dir/$filename";
while ( <$upload_filehandle> )
{
            print UPLOADFILE;
}
close UPLOADFILE;
```

The file uploading works with a special type of form field called "file" and form encoding called "multipart/form-data".

1) Use the perl CGI library.
   Upload directory : The location in the server where to store the uploaded files.
   ie a directory under the document root.
   $upload_dir = "C:/xampp/htdocs/Trials/upload/"; The absolute path to that directory is specified.
2) Reading the form variables: Read the file name of the uploaded file.
   $filename = $query->param("photo");
3) Some browsers pass the whole path to the file, instead of the filename alone so strip off everything that includes backslashes(for windows browsers) and forward slashes(for unix browsers) and which might appear before the filename .
   $filename =~ s/.*[\/\\](.*)/$1/;
4) Get the file handle:
   Upload method (upload()) to get the file handle of the uploaded file. The file handle points to a temporary file created by CGI.pm module.
   $upload_filehandle = $query->upload("photo");
5) Saving the file:
   The file handle to the uploaded file is used to read its contents and save it out to a new file in the destination location. Use the uploaded file's filename as the name of the new file

```
  open UPLOADFILE, ">$upload_dir/$filename";
  while ( <$upload_filehandle> )
   {
       print UPLOADFILE;
```

7	a.	Explain CGI Scripting . Explain CGI.pm methods.	[10]	CO3	L4

Common Gateway Interface(CGI) is a standard way for web servers to interface with executable programs installed on a server that generate web pages dynamically. Such programs are known as CGI scripts . They are usually written in a scripting language. Each Web server runs HTTP server software, which responds to requests from Web browsers. Generally, the HTTP server has a directory (folder), which is designated as a document collection — files that can be sent to Web browsers connected to this server .

CGI.pm is a library of routines that simplify the creation and processing of html web forms. It has two aspects:
The processing of data returned from the client browsers and the dynamic creation of html pages containing web forms. The ability to easily extract values from returned data and create dynamic web forms gives the developer a simple way to maintain state across the web. The CGI.pm module can safely handle GET, POST and multipart MIME data to extract data from the web forms. The CGI.pm module can be used in a simple functional programming style or in an object oriented way.

```
#!/usr/bin/per
use CGI ":standard";
$q = new CGI;
print $q->header;
print $q->start_html("welcome");
print $q->h1("hello");
print $q->end_html();
```

This program can be saved in C:\xampp\htdocs\program.cgi and execute as localhost\program.cgi

Ex: 2

```
#!"C:\xampp\perl\bin\perl.exe"

use CGI qw/:standard/;
use CGI::Carp(fatalsToBrowser);
use strict;

my $page = new CGI;

print ($page->header(),$page->start_html("parameters"),$page->h1("params
and values"));

my @param_names = $page->param;
my $next;
print "<ul>";

foreach $next(@param_names)
{
        print "<li>".$next."=>".$page->param($next)."</li>";
}
print "</ul";
print $page->end_html();
print "\n";

exit(0);
```

8	a.	How to create and destroy a cookie in PERL.	[05]	CO3	L4

A cookie is a small piece of information that is stored on the client machine . They can be used to restrict access to whole areas, can be set to expire so that they provide a simple form of access control and provide lots of information.

HTTP protocol is a stateless protocol. But for a commercial website it is required to maintain session information among different pages. For example one user registration ends after completing many pages. But how to maintain user's session information across all the web pages.

In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

How It Works

Your server sends some data to the visitor's browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives at another page on your site, the cookie is available for retrieval. Once retrieved, your server knows/remembers what was stored.

Cookies are a plain text data record of 5 variable-length fields:

- **Expires :** The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.
- **Domain :** The domain name of your site.
- **Path :** The path to the directory or web page that set the cookie. This may be blank if you want to retrieve the cookie from any directory or page.
- **Secure :** If this field contains the word "secure" then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.
- **Name=Value :** Cookies are set and retrviewed in the form of key and value pairs.

Ex **to set a cookie**

```
#!"C:\xampp\perl\bin\perl.exe"

use CGI qw/:standard/;
use strict;
use CGI::Carp(fatalstoBrowser);

my %txtval = ('visit'=>'1');
my $cookie = cookie(-name=>'cname',-value=>\%txtval,-path=>'/',-
expire=>'+2h');

print header(-cookie=>$cookie);
print start_html('creating a cookie');
print h1('creating cookies');
print end_html();

exit(0);
```

Ex **to delete a cookie**

```
#!"C:\xampp\perl\bin\perl.exe"

use CGI qw/:standard/;
use strict;
use CGI::Carp(fatalstoBrowser);

my $cookie = cookie(-name=>'cname',-value=>' ',-path=>'/',-expire=>'-2h');
```

```
print header(-cookie=>$cookie);
print start_html('creating a cookie');
print h1('creating cookies');
print end_html();

exit(0);
```

b. What are References in PERL . Explain in detail. [05] CO3 L4

A Perl reference is a scalar data type that holds the location of another value which could be scalar, arrays, or hashes. Because of its scalar nature, a reference can be used anywhere, a scalar can be used. It is easy to create a reference for any variable, subroutine or value by prefixing it with a backslash as follows –

```
$scalarref = \$foo;
$arrayref  = \@ARGV;
$hashref   = \%ENV;
```

Dereferencing returns the value from a reference point to the location. To dereference a reference simply use $, @ or % as prefix of the reference variable depending on whether the reference is pointing to a scalar, array, or hash. Following is the example to explain the concept –

```
#!/usr/bin/perl

$var = 10;
# Now $r has reference to $var scalar.
$r = \$var;
# Print value available at the location stored in $r.
print "Value of $var is : ", $$r, "\n";
@var = (1, 2, 3);
# Now $r has reference to @var array.
$r = \@var;
# Print values available at the location stored in $r.
print "Value of @var is : ",  @$r, "\n";
%var = ('key1' => 10, 'key2' => 20);
# Now $r has reference to %var hash.
$r = \%var;
# Print values available at the location stored in $r.
print "Value of %var is : ", %$r, "\n";
```

Course Outcomes		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8
CO1:	Develop Web apps using various development languages and tools	1	-	3	-	-	-	3	3
CO2:	Build the ability to select the essential technology needed to develop and implement web applications	2	2		-	-	1	2	3
CO3:	Design dynamic web applications using PERL CGI - MySQL		3	3	1	-	1	3	3
CO4:	Design dynamic web applications using PHP MySQL	-	-	3	2	-	-	3	3
CO5:	Ruby Rails application development	1	-	2	-	-	-	3	3
CO6:	Develop Web apps using various development languages and tools	-	-	-	1	2	2	-	-

Cognitive level	KEYWORDS
L1	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.
L2	summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
L3	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover.
L4	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer.
L5	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize.