USN										

CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test 2 – Nov. 2017

Sub:	Programming the Web					Sub Code:	10CS73	Branch:	CSE	
Date:	08-11-2017	Duration:	90 min's	Max Marks:	50	Sem / Sec:	7 – A, B , C			OBE

Answer any 5 questions

		Marks	CO	RBT
1.a)	Explain the different primitive types in javascript.	3	CO1	L2
b)	Explain the screen output and keyboard input methods, with examples.	3	CO1	L2
c)	Write a Javascript that contains a function named validate - phoneno, which tests the phone number of the format ddd - dddd - ddddddd <09-8256-1234567> and display whether the given number is valid or not using alert.	4	CO2	L3
2. a)	Write a javascript script that checks the passwords, that includes two passwords as input element, along with reset and submit buttons. Implement the below mentioned functions to check: i) Both entered passwords are same. ii) Both entered passwords are different. iii) If no password is typed in either of the password fields. iv) Use on submit to trigger a call to display an alert box if the error occurs.	5	CO2	L3
b)	Explain object creation and modification in JavaScript.	5	CO1	L2
3.a)	Briefly discuss the event handling from body elements and button elements in javascript.	10	CO3	L2
4.a)	Explain the different types of positioning elements with example.	6	CO3	L2
b)	Write a javascript to validate the name, the name should be entered using prompt. The first and last name should not more than 10 characters and middle name must contain only initial. If so display validation corresponding to name. The format is first_name Second_name third_name. There should be single white space between first_name second_name and third_name.	4	CO2	L3
5.a)	Explain the three phases of event processing in the DOM2 event model.	5	CO3	L2
b)	Describe the approach to addressing XHTML elements using forms and elements.	5	CO3	L2
6.a)	Describe built-in list functions in perl.	4	CO4	L2
b)	Write a program to read a file on command line that contains person's name, in each line convert them into uppercase and displays them in ascending order.	4	CO4	L3
c)	Explain remembering matches in perl.	2	CO4	L2
7. a)	Explain the query string format and cgi.pm module	5	CO4	L2
b)	Write a Perl program which creates a hash table contain country names keys and their capitals as values and perform the following: i) Print all pairs of values (country names and capital). ii) Accept country name and print the capital of it.	5	CO4	L3
8.a)	Explain pattern matching in perl.	7	CO4	L2
b)	Write a Perl program to copy contents of one file to another.	3	CO4	L3

1.a) Explain the different primitive types in javascript. (3)

The types can be divided into two groups: primitive types and reference types. Numbers, boolean values, and the null and undefined types are primitive. Objects, arrays, and functions are reference types.

A primitive type has a fixed size in memory. For example, a number occupies eight bytes of memory, and a boolean value can be represented with only one bit. The number type is the largest of the primitive types. If each JavaScript variable reserves eight bytes of memory, the variable can directly hold any primitive value.

Reference types are another matter, however. Objects, for example, can be of any length -- they do not have a fixed size. The same is true of arrays: an array can have any number of elements. Similarly, a function can contain any amount of JavaScript code. Since these types do not have a fixed size, their values cannot be stored directly in the eight bytes of memory associated with each variable. Instead, the variable stores a *reference* to the value. Typically, this reference is some form of pointer or memory address. It is not the data value itself, but it tells the variable where to look to find the value.

```
var a = 3.14;  // Declare and initialize a variable
var b = a;     // Copy the variable's value to a new variable
a = 4;         // Modify the value of the original variable
alert(b)       // Displays 3.14; the copy has not changed
```

1. b) Explain the screen output and keyboard input methods, with examples. (3)

- The JavaScript model for the HTML document is the Document object
- The model for the browser display window is the Window object
  - The Window object has two properties, document and window, which refer to the document and window objects, respectively
- The Document object has a method, write, which dynamically creates content
  - The parameter is a string, often catenated from parts, some of which are variables
    e.g., document.write("Answer: " + result + "
");
- The Window object has three methods for creating dialog boxes, alert, confirm, and prompt
  1. alert("Hej! \n");
     - Parameter is plain text, not HTML
     - Opens a dialog box which displays the parameter string and an OK button
  2. confirm("Do you want to continue?");
     - Opens a dialog box and displays the parameter and two buttons, OK and Cancel
  3. prompt("What is your name?", "");
     - Opens a dialog box and displays its string parameter, along with a text box and two buttons, OK and Cancel
     - The second parameter is for a default response if the user presses OK without typing a response in the text box (waits for OK)

1.c) Write a Javascript that contains a function named validate - phoneno, which tests the phone number of the format ddd - dddd - ddddddd <09-8256-1234567> and display whether the given number is valid or not using alert. (4)

```
<html>
	<head>
		<title>q.1b</title>
	</head>
	<body>
		<script type="text/javascript">
			var a=prompt("Enter the phone number","");
			var pat=/\d{3}-\d{4}-\d{7}/;
			if(a.match(pat))
```

```
                {
                        alert("phone number matched");
                }
                else
                {
                        alert("phone number is not matched");
                }
        </script>
    </body>
</html>
```

2.a)

Write a javascript script that checks the passwords, that includes two passwords as input element, along with reset and submit buttons. Implement the below mentioned functions to check:

i) Both entered passwords are same.

ii) Both entered passwords are different.

iii) If no password is typed in either of the password fields.

iv) Use on submit to trigger a call to display an alert box if the error occurs.

```
<html>
    <head>
        <title>q.2a</title>
        <script type="text/javascript">
            function chk()
            {
                var a1=document.getElementById("p1").value;
                var a2=document.getElementById("p2").value;
                if(a1=="" || a2=="")
                {
                    alert("required fields are empty");
                }
                else if(a1==a2)
                {
                    alert("Password are same");
                }
                else if(a1!=a2)
                {
                    alert("Password are different");
                }

            }
        </script>
    </head>
    <body>
        <form action="" method="" onsubmit="chk()">
            Enter password 1:<input type="password" id="p1" />
            Enter password 2:<input type="password" id="p2" />
            <input type="submit" value="Check" />
            <input type="reset" value="Clear" />
        </form>
    </body>
</html>
```

2.b) Explain object creation and modification in JavaScript. (5)

A constructor is a function that creates and initializes an object. JavaScript provides a special constructor function called **Object()** to build the object. The return value of the **Object()** constructor is assigned to a variable.

A JavaScript object has properties associated with it. A property of an object can be explained as a variable that is attached to the object. Object properties are basically the same as ordinary JavaScript variables, except for the attachment to objects. The properties of an object define the characteristics of the object. You access the properties of an object with a simple dot-notation:

```
objectName.propertyName
```

Like all JavaScript variables, both the object name (which could be a normal variable) and property name are case sensitive. You can define a property by assigning it a value. For example, let's create an object named myCar and give it properties named make, model, and year as follows:

```
var myCar = new Object();
myCar.make = 'Ford';
myCar.model = 'Mustang';
myCar.year = 1969;
```

Unassigned properties of an object are undefined (and not null).

```
myCar.color; // undefined
```

Properties of JavaScript objects can also be accessed or set using a bracket notation (for more details see property accessors). Objects are sometimes called *associative arrays*, since each property is associated with a string value that can be used to access it. So, for example, you could access the properties of the myCar object as follows:

```
myCar['make'] = 'Ford';
myCar['model'] = 'Mustang';
myCar['year'] = 1969;
```

3.a) Briefly discuss the event handling from body elements and button elements in javascript(5)

Onload

```
<html>
	<head>
		<title>q.2a</title>
		<script type="text/javascript">
			function ol()
			{
				document.write("message from body");
			}
		</script>
	</head>
	<body onload="ol()">
		<p>message from window</p>
	</body>
</html>
```

Button elements

```
<!DOCTYPE html>
<html>
<body>

<button onclick="myFunction()">Click me</button>

<p id="demo"></p>

<p>A function is triggered when the button is clicked. The function outputs some text in a p element with id="demo".</p>

<script>
function myFunction() {
  document.getElementById("demo").innerHTML = "Hello World";
}
```

```
</script>

</body>
</html>
```

4.a) Explain the different types of positioning elements with example.(6)

static

This is the default value and specifies that the element is positioned according to the normal flow of document content (for most Western languages, this is left to right and top to bottom.) Statically positioned elements are not DHTML elements and cannot be positioned with the top, left, and other attributes. To use DHTML positioning techniques with a document element, you must first set its position attribute to one of the other three values.

absolute

This value allows you to specify the position of an element relative to its containing element. Absolutely positioned elements are positioned independently of all other elements and are not part of the flow of statically positioned elements. An absolutely positioned element is positioned either relative to the <body> of the document or, if it is nested within another absolutely positioned element, relative to that element. This is the most commonly used positioning type for DHTML.

```
<div style="position: absolute; left: 100px; top: 100px;">
```

relative

When the position attribute is set to relative, an element is laid out according to the normal flow, and its position is then adjusted relative to its position in the normal flow. The space allocated for the element in the normal document flow remains allocated for it, and the elements on either side of it do not close up to fill in that space, nor are they "pushed away" from the new position of the element. Relative positioning can be useful for some static graphic design purposes, but it is not commonly used for DHTML effects.

4.b) Write a javascript to validate the name, the name should be entered using prompt. The first and last name should not more than 10 characters and middle name must contain only initial. If so display validation corresponding to name. The format is first_name Second_name third_name. There should be single white space between first_name second_name and third_name. (4)

```
<html>
	<head>
		<title>q.1b</title>
	</head>
	<body>
		<script type="text/javascript">
			var a=prompt("Enter the first name","");
			var b=prompt("Enter the middle name","");
			var c=prompt("Enter the last name","");
			var pat1=/[a-zA-Z]{1,10}/;
			var pat2=/[a-zA-Z]{1}/;
			var pat3=/[a-zA-Z]{1,10}/;
			if(a.match(pat1) && b.match(pat2) && c.match(pat3))
			{
				document.write("The name is: "+a+" "+b+" "+" "+c);
			}
			else
			{
				document.write("wrong pattern");
			}
		</script>
	</body>
</html>
```

5. a) Explain the three phases of event processing in the DOM2 event model. (5)

1. The capture phase: the event is dispatched to the target's ancestors from the root of the tree to the direct parent of the target node.
2. The target phase: the event is dispatched to the target node.
3. The bubbling phase: the event is dispatched to the target's ancestors from the direct parent of the target node to the root of the tree.

5.b) Describe the approach to addressing XHTML elements using forms and elements. (5)

Javascript provides the ability for getting the value of an element on a webpage as well as dynamically changing the content within an element. Getting the value of an element To get the value of an element, the **getElementById** method of the **document** object is used. For this method to get the value of an element, that element has to have an id given to it through the **id** attribute

Changing the content within an element To change the content within an element, use the innerHTML property. Using this property, you could replace the text in paragraphs, headings and other elements based on several things such as a value the user enters in a textbox. For this property to change the content within an element, that element has to have an 'id' given to it through the **id** attribute.

6.a) Describe built-in list functions in perl.(4)

- Sort()

  @a=array(3,2,5,1);
  @b=sort @a;
- Split()
  - $a=”here I am”;
  - @s=split(“ “,$a);
- Die

  Die “error”;
- qw

  qw(apple orange grape);

6.b) Write a program to read a file on command line that contains person’s name, in each line convert them into uppercase and displays them in ascending order.

```
$index = 0;
#>>> Loop to read the names and process them
while($name = < >) {
#>>> Convert the name's letters to uppercase and put it in the names array. $names[$index++] = uc($name);
}
#>>> Display the sorted list of names
print  “\nThe Sorted list of names is:\n\n\n”;
foreach $name (sort @names)
{
print (“$name \n”);
}
```

6 . c) Explain remembering matches in perl.

```
$s1=”4 Aug 1947”;
$s1=~/(\d+) (\w+) (\d+) /;
Print $1 $2 $3;
```

7.a) Explain the query string format and cgi.pm module

A query string is the part of a URL which is attached to the end, after the file name. It begins with a question mark and usually includes information in pairs. The format is **parameter=value**, as in the following example:

www.mediacollege.com/cgi-bin/myscript.cgi?topic=intro

Query strings can contain multiple sets of parameters, separated by an ampersand (**&**) like so:

www.mediacollege.com/cgi-bin/myscript.cgi?topic=intro&heading=1

The idea is that the information contained in the query string can be accessed and interpreted by a script. Specifically, the query string data is contained in the variable **$ENV{'QUERY_STRING'}**.

CGI.pm module

```
#!/usr/bin/perl -w
# cgi script with CGI.pm

use CGI;

$query = CGI::new();
$bday = $query->param("birthday");
print $query->header();
print $query->p("Your birthday is $bday.");
```

Even for this tiny program, you can see that CGI.pm can alleviate many of the headaches associated with CGI programming.

A script to create a fill-out (cgipm.pl) form that remembers its state each time it's invoked is very easy to write with CGI.pm:

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

use CGI qw(:standard);

print header;
print start_html('A Simple Example'),
   h1('A Simple Example'),
   start_form,
   "What's your name? ",textfield('name'),
   p,
   "What's the combination?",
   p,
   checkbox_group(-name=>'words',
                  -values=>['eenie','meenie','minie','moe'],
                  -defaults=>['eenie','minie']),
   p,
   "What's your favorite color? ",
   popup_menu(-name=>'color',
              -values=>['red','green','blue','chartreuse']),
   p,
   submit,
   end_form,
   hr;

if (param()) {
   print
      "Your name is",em(param('name')),
      p,
      "The keywords are: ",em(join(", ",param('words'))),
      p,
```

```
        "Your favorite color is ",em(param('color')),
        hr;
}
print end_html;
```

7.b) Write a Perl program which creates a hash table contain country names keys and their capitals as values and perform the following:
i) Print all pairs of values (country names and capital).
ii) Accept country name and print the capital of it.

```
%capitals = ('china' => 'beijing', 'england' => 'london', 'france' => 'paris', 'norway' => 'oslo', 'italy' => 'rome');

 # process hash elements
# write to file in csv format
foreach $k (keys (%capitals))
{
        print FILE $k, ",", $capitals{$k}, "\n";
}
 $search=<STDIN>;
if(exists  $capitals{$search})
            print $capitals{$search};
```

8.a) Explain pattern matching in perl. [7]

Patterns

Patterns are subject to an additional level of interpretation as a regular expression. This is done as a second pass, after variables are interpolated, so that regular expressions may be incorporated into the pattern from the variables. If this is not what you want, use \Q to interpolate a variable literally.

?PATTERN?

This is just like the /pattern/ search, except that it matches only once between calls to the reset() operator. This is a useful optimization when you only want to see the first occurrence of something in each file of a set of files, for instance. Only ?? patterns local to the current package are reset.

m/PATTERN/gimosx

Searches a string for a pattern match, and in a scalar context returns true (1) or false ('). If no string is specified via the =~ or !~ operator, the $_ string is searched. The string specified with =~ can be a variable or the result of an expression evaluation. The initial 'm' can be omitted if '/' is used for the delimiters, otherwise any non-alphanumeric character can be used (apart from whitespace).

The modifier options are:

g	Match globally, i.e. find all occurrences.
i	Do case-insensitive pattern matching.
m	Treat string as multiple lines - default is to assume just a single line in the string (no embedded newlines). See $*.
o	Only compile pattern once, even if variables within it change.
s	Treat string as single line.
x	Use extended regular expressions. Whitespace that is not backslashed or within a haracter class is ignored, allowing the regular expression to be broken into more readable parts with embedded comments.

In a list context, the pattern match returns the portions of the target string that match the expressions within the pattern in brackets. In a scalar context, each iteration identifies the next match (pos() holding the position of the previous match on the variable).

q/STRING/, 'STRING'

A single-quoted, literal string, default delimiters are single quotes ('...'). Backslashes are ignored, unless followed by the delimiter or another backslash, in which case the delimiter or backslash is interpolated.

qq/STRING/, "STRING"

A double-quoted, interpolated string, default delimiters are double quotes ("...").

qx/STRING/, `STRING`

A string which is interpolated and then executed as a system command. The collected standard output of the command is returned. In scalar context, it comes back as a single (potentially multi-line) string. In list context, returns a list of lines (depending on how the $/ delimiter is specified).

```
$today = qx{ date };
```

qw/STRING/

Returns a list of the words extracted out of STRING, using embedded whitespace as the word delimiters. It is exactly equivalent to:

```
split(' ', q/STRING/);
```

Some frequently seen examples:

```
use POSIX qw( setlocale localeconv )
@EXPORT = qw( foo bar baz );
```

s/PATTERN/REPLACEMENT/egimosx

Searches a string for a pattern, and if found, replaces that pattern with the replacement text and returns the number of substitutions made. Otherwise it returns false (0).

If no string is specified via the =~ or !~ operator, the $_ variable is searched and modified. (The string specified with =~ must be a scalar variable, an array element, a hash element, or an assignment to one of those, i.e. an lvalue.)

If the delimiter chosen is single quote, no variable interpolation is done on either the PATTERN or the REPLACEMENT. Otherwise, if the PATTERN contains a $ that looks like a variable rather than an end-of-string test, the variable will be interpolated into the pattern at run-time. If you only want the pattern compiled once the first time the variable is interpolated, use the /o option. If the pattern evaluates to a null string, the most recently executed (and successfully compiled) regular expression is used instead.

The modifier options are (see m/PATTERN/ above for more detailed descriptions of common modifiers):

e	Evaluate the right side as an expression.
g	Match globally, i.e. all occurrences.
i	Case-insensitive pattern matching.
m	Treat string as multiple lines.
o	Only compile pattern once, even if variables within it change.
s	Treat string as single line.
x	Use extended regular expressions

Any non-alphanumeric, non-whitespace delimiter may replace the slashes. If single quotes are used, no interpretation is done on the replacement string, overwridden by the /e modifier. If backquotes are used, the replacement string is a command to execute whose output will be used as the actual replacement text. If the PATTERN is delimited by bracketing quotes, the REPLACEMENT has its own pair of quotes, which may or may not be bracketing quotes, e.g. s(foo)(bar) or s<foo>/bar/ . A /e will cause the replacement portion to be interpreter as a full-fledged Perl expression and eval() ed right then and there. It is, however, syntax checked at compile-time.

Examples:

```
s/\bgreen\b/mauve/g;                # don't change wintergreen
$path =~ s|/usr/bin|/usr/local/bin|;
s/Login: $foo/Login: $bar/; # run-time pattern
($foo = $bar) =~ s/this/that/;
$count = ($paragraph =~ s/Mister\b/Mr./g);
$_ = 'abc123xyz';
s/\d+/$&*2/e;               # yields 'abc246xyz'
s/\d+/sprintf("%5d",$&)/e;  # yields 'abc  246xyz'
s/\w/$& x 2/eg;             # yields 'aabbcc  224466xxyyzz'
s/%(.)/$percent{$1}/g;      # change percent escapes; no /e
s/%(.)/$percent{$1} || $&/ge;       # expr now, so /e
s/^=(\w+)/&pod($1)/ge;      # use function call
# /e's can even nest;  this will expand
# simple embedded variables in $_
s/(\$\w+)/$1/eeg;
```

```
# Delete C comments.
$program =~ s {
    /\*     (?# Match the opening delimiter.)
    .*?     (?# Match a minimal number of characters.)
    \*/     (?# Match the closing delimiter.)
} []gsx;
s/^\s*(.*?)\s*$/$1/;        # trim white space
s/([^ ]*) *([^ ]*)/$2 $1/;  # reverse 1st two fields
```

Occasionally, you can't just use a /g to get all the changes to occur. Here are two common cases:

```
# put commas in the right places in an integer
1 while s/(.*\d)(\d\d\d)/$1,$2/g;      # perl4
1 while s/(\d)(\d\d\d)(?!\d)/$1,$2/g;  # perl5
# expand tabs to 8-column spacing
1 while s/\t+/' ' x (length($&)*8 - length($`)%8)/e;
```

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

Translates all occurrences of the characters found in the search list with the corresponding character in the replacement list. It returns the number of characters replaced or deleted. If no string is specified via the =~ or !~ operator, the $_ string is translated. (The string specified with =~ must be a scalar variable, an array element, or an assignment to one of those, i.e. an lvalue.) For sed devotees, y is provided as a synonym for tr . If the SEARCHLIST is delimited by bracketing quotes, the REPLACEMENTLIST has its own pair of quotes, which may or may not be bracketing quotes, e.g. tr[A-Z][a-z] or tr(+-*/)/ABCD/ .

Options are:

c	Complement the SEARCHLIST.
d	Delete found but unreplaced characters.
s	Squash duplicate replaced characters.

If the /c modifier is specified, the SEARCHLIST character set is complemented. If the /d modifier is specified, any characters specified by SEARCHLIST not found in REPLACEMENTLIST are deleted. (Note that this is slightly more flexible than the behavior of some tr programs, which delete anything they find in the SEARCHLIST, period.) If the /s modifier is specified, sequences of characters that were translated to the same character are squashed down to a single instance of the character.

If the /d modifier is used, the REPLACEMENTLIST is always interpreted exactly as specified. Otherwise, if the REPLACEMENTLIST is shorter than the SEARCHLIST, the final character is replicated till it is long enough. If the REPLACEMENTLIST is null, the SEARCHLIST is replicated. This latter is useful for counting characters in a class or for squashing character sequences in a class.

Examples:

```
$ARGV[1] =~ tr/A-Z/a-z/;    # canonicalize to lower case
$cnt = tr/*/*/;             # count the stars in $_
$cnt = $sky =~ tr/*/*/;     # count the stars in $sky
$cnt = tr/0-9//;            # count the digits in $_
tr/a-zA-Z//s;               # bookkeeper -> bokeper
($HOST = $host) =~ tr/a-z/A-Z/;
tr/a-zA-Z/ /cs;             # change non-alphas to single space
tr [\200-\377]
   [\000-\177];             # delete 8th bit
```

Note that because the translation table is built at compile time, neither the SEARCHLIST nor the REPLACEMENTLIST are subjected to double quote interpolation. That means that if you want to use variables, you must use an eval():

```
eval "tr/$oldlist/$newlist/";
die $@ if $@;
eval "tr/$oldlist/$newlist/, 1" or die $@;
```

Regular Expressions

The patterns used in pattern matching are regular expressions that follow the rules laid out below.

Any single character (or series of characters) matches directly, unless it is a metacharacter with a special meaning.

You can cause characters which normally function as metacharacters to be interpreted literally by prefixing them with a "\" (e.g. "\." matches a ".", not any character; "\\" matches a "\"). A series of characters matches that series of characters in the target string, so the pattern *zyxwv* would match "zyxwv" in the target string.

The following metacharacters are as supported:

\	Quote the next metacharacter, including escape sequences (\n, \t etc. apart from \b - see below), ASCII characters ('\nnn' for octal and '\xnn' for hex), and ASCII character controls ('\cx'). '\ n' repeats the part of the 'n'th subpattern that was used to perform the match (not its complete set of rules).
^	Match just the beginning of the string, or with the /m modifier the beginning of any embedded line
.	Match any character (except newline unless the /s modifier is used)
$	Match just the end of the string, or with the /m modifier the end of any embedded line
|	Alternation - to match any one of a set of patterns, usually grouped in brackets.
()	Grouping of subpatterns, numbered automatically left to right by the sequence of their opening parenthesis.
[]	Character class, matching any of the characters in the enclosed list. '^' as the first character in the list negates the expressions - any character *not* in the list.

The following quantifiers are suppported:

*	Match 0 or more times (equivalent to {0,})
+	Match 1 or more times (equivalent to {1,})
?	Match 0 or 1 times (equivalent to {0,1})
{n}	Match exactly n times
{n,}	Match at least n times
{n,m}	Match at least n but not more than m times

Patterns that are qualified as above match as many times as possible without causing the rest of the match to fail, by default. To match the fewest number of times (to ensure that multiple matches of the super-pattern are found) suffix the quantifier with '?', eg. '*?', '{n,m}?'.

Regular expressions also support the following constructs:

Single character matches		Zero width matches	
\w	a "word" character (alphanumeric plus "_")	\b	a word boundary
\W	a non-word character	\B	a non-(word boundary)
\s	a whitespace character	\A	beginning of the string (not embedded newlines)
\S	a non-whitespace character	\Z	end of the string (not embedded newlines)
\d	a digit character	\G	where previous m//g left off
\D	a non-digit character		

Brackets delimit sub-patterns, allowing the resultant matches in the target string to be referenced using either /1 ... /n within the pattern itself, or $1 ... $n outside of the pattern. If the '(' is followed by a '?', it can be used to delimit a subpattern without the pattern being saved.

$+	the last pattern that was matched (useful when there are alternatives)
$&	the matched string
$`	everything before the matched string
$'	everything after the matched string

The extension syntax for regular expressions uses a pair of brackets where the first character within the brackets is a question mark '(?...)'.

(?#comment) A comment - ignored

(?:regexp) Groups the pattern as with brackets, but doesn't generate back references

(?=regexp) A zero-width positive lookahead assertion. For example, /\w+(?=\t)/ matches a word followed by a tab, without including the tab in $&.

(?!regexp) A zero-width negative lookahead assertion. For example:

```
/abc(?!xyz)/
```

matches any occurrence of "abc" that isn't followed by "xyz". This cannot be used for lookbehind: /(?!abc)xyz/ will not find an occurrence of "xyz" that is preceded by something which is not "abc". The (?!abc) is ensuring that the next thing is not "abc" - and it's not, it's "xyz", so "abcxyz" will match. Would have to do something like /(?abc)...xyz/ for that, but there's the case where the "xyz" does not have three characters before it. This could be covered by:

```
/(?:(?!abc)...|^..?)xyz/
```

It may be easier to say:

```
if (/abc/ && $` =~ /xyz$/)
```

(?imsx) One or more embedded pattern-match modifiers. Useful for patterns that are specified in a table somewhere, some of which want to be case sensitive, and some of which don't. The case insensitive ones merely need to include (?i) at the front of the pattern. For example:

```
$pattern = "abcxyz";
if ( /$pattern/i )
# more flexible:
$pattern = "(?i)abcxyz";
if ( /$pattern/ )
```

8.b) Write a Perl program to copy contents of one file to another.[3]

```
use vars qw($filecontent $total);
my $file1 = "file1.txt";open(FILE1, $file1) || die "couldn't open the file!";
open(FILE2, '>>file2.txt') || die "couldn't open the file!";
while($filecontent = <FILE1>)
{
chomp($filecontent);
print FILE2 $filecontent."\n"
}
```