FILE TEST OPERATORS

```
if (-operator 'filename') { print "filename exists!\n"; }

if (-e 'filename') { print "filename exists!\n"; }

if (!(-e 'filename')) { open ">filename"; }
```

Operator	Test
-d	if filename is a directory
-e	if file exists
-s	if file is non-empty
-z	if file is empty
-r	if file is a readable file
-f	if file is a plain file
-T	if file a text file
-w	if file is writable
-x	if file is executable
-B	if file is binary

```
open(IN, $file) or die "Error opening $file: $!\n";
while($line = <IN>) { print "$line\n";}
```

DEBUGGING

The Perl debugger is invoked with the –d option:

```
% perl –d script_name.pl
```

- h or h h – for help page
- c – to continue down from current execution till the breakpoint otherwise till the subroutine name or line number,
- p – to show the values of variables,
- b – to place the breakpoints,
- L – to see the breakpoints set,
- d – to delete the breakpoints,
- s – to step into the next line execution.
- n – to step over the next line execution, so if next line is subroutine call, it would execute subroutine but not descend into it for inspection.
- source file – to take the debug commands from the file.
- l subname – to see the execution statements available in a subroutine.
- q – to quit from the debugger mode.

/string/		do a forward string search in d program
?string?		Do a reverese string search
S		dispays all sub routies in program one per line
f		execute d reminder of a sub routine
w n		display lines sorrounding line n
D		all break points
	-	Display lines b4 d current line
	l	list d next few lines of code vth line no
	l n+m	display m lines starting frm nth line
	l n-m	display lines starting frm m to n
	l sub	display code vthin sub routine
	L	list all break points

REGULAR EXPRESSIONS

- // [pattern delimiter]
- =~ [binding operator]
- !~ [negation operator]
- s/// [substitution operator]
- tr/// [translation operator]

```
$gene =~ /ribonuclease/;      #default (m not required)
$gene =~ m!ribonuclease!;     #m required
$gene =~ m#ribonuclease#;     #m required
$gene =~ m%ribonuclease%;     #m required
```

■ g	[enable global substitutions
■ i	[enable case-insensitive matches
■ |	[enable matching a list of patterns in ()
■ m	[treat pattern as multiple lines
■ s	[treat pattern as a single line

■ ?=	[conditional positive matching
■ ?!	[conditional negative matching

[0-9]	[matches any single digit
[a-z]	[matches any single alphabet
[a-z0-9]	[matches a single digit or alphabet in any order
[;:]	[matches a single space or ; or :

ESCAPE CHARACTERS

.	[matches any single character except \n
\d	[matches any single digit
\D	[matches any single non-digit
\s	[matches any single space
\S	[matches any single non-white space
\w	[matches any single word
\W	[matches any single non-word
[x..y]	[specifies a range of patterns

CHARACTER SET

[0-9]	means match a single digit,
[^0-9]	means match a single non-digit

\d	[0-9]	[any single digit]
\D	[^0-9]	[any single non-digit]
\s	[\n\t\r\f]	[white space]
\S	[^ \n\t\r\f]	[any single non-white space]
\w	[_0-9a-zA-Z]	[any single word character]
\W	[^_0-9a-zA-Z]	[any single non-word character]

- + [match one or more instances of a given pattern
- * [match zero or more instances of a given pattern
- ? [match zero or one instance of a given pattern

Quantifier	Definition
+	{1, }
*	{0, }
?	{0,1}

/ez{1,5}/ matches ez, ezz, ezzz, ezzzz and ezzzzz

ANCHORS

- ^ or \A [match at beginning of string]
- $ or \Z [match at end of string]
- \b [match at beginning or end of word]
- \B [match only inside a word]

OPTIONS OF SUBSTITUTION OPERATOR

- x [ignore white space in pattern]
- m [enable matching over multiple-lines]
- e [evaluate substitution string as expression]

OPTIONS OF TRANSLATION OPERATOR

- c [translate all characters that are not specified]
- d [delete all specified characters]
- s [replace identical output characters with a single character]

/bkw/	**matches only kw present at beginning of a word**
/kw\b/	**matches only kw present at end of a word**
/\bkw\b/	**matches exactly kw**
/\Bkw/	**match only kw present at the end or within d word**

Definition of CGI

CGI is the *Common Gateway Interface*, a standard for programs to interface with information servers such as HTTP (web) servers. CGI allows the HTTP server to run an executable program or script in response to a user request, and generate output on the fly. This allows web developers to create dynamic and interactive web pages.

CGI programs can be written in any language. Perl is a very common language for CGI programming as it is largely platform independent and the language's features make it very easy to write powerful applications. However, some CGI programs are written in C, shell script, or other languages.

It is important to remember that CGI is not a language in itself. CGI is merely a type of program which can be written in any language.

A simple HTTP transaction, such as a request for a static HTML page, works as follows:

1. The user types a URL into his or her browser, or specifies a web address by some other means such as clicking on a link, choosing a bookmark, etc
2. The user agent connects to port upon which the HTTP server is running (usually port 80)
3. The user agent sends a request such as `GET /index.html`
4. The user agent may also send other headers
5. The HTTP server receives the request and finds the requested file in its filesystem
6. The HTTP server sends back some HTTP headers, followed by the contents of the requested file
7. The HTTP server closes the connection

When a user requests a CGI program, however, the process changes slightly:

1. The user agent sends a request as above
2. The HTTP server receives the request as above
3. The HTTP server finds the requested CGI program in its file system
4. The HTTP server executes the program
5. The program produces output, including HTTP headers
6. The HTTP server sends back the output of the program
7. The HTTP server closes the connection

What is needed to run Perl CGI programs?

There are several things you need in order to create and run Perl CGI programs.

- a web server
- web server configuration which gives you permission to run CGI
- a Perl interpreter
- appropriate Perl modules, such as CGI.pm
- a shell account is extremely useful but not essential

Most of the above requirements will need your system administrator or ISP to set them up for you. Some will be wary of allowing users to run CGI programs, and may require you to obey certain security regulations or pay extra for the privilege.

BIOPERL

Bio::Perl has a number of other easy-to-use functions, including

```
get_sequence        - gets a sequence from standard, internet accessible
                          databases
read_sequence       - reads a sequence from a file
read_all_sequences  - reads all sequences from a file
new_sequence        - makes a Bioperl sequence just from a string
write_sequence      - writes a single or an array of sequence to a file
translate           - provides a translation of a sequence
translate_as_string - provides a translation of a sequence, returning back
                      just the sequence as a string
blast_sequence      - BLASTs a sequence against standard databases at
                      NCBI
write_blast         - writes a blast report out to a file
```

Using Bioperl

Bioperl provides software modules for many of the typical tasks of bioinformatics programming. These include:

- Accessing sequence data from local and remote databases
- Transforming formats of database/ file records
- Manipulating individual sequences
- Searching for similar sequences
- Creating and manipulating sequence alignments
- Searching for genes and other structures on genomic DNA
- Developing machine readable sequence annotations

Accessing sequence data from local and remote databases

```
$seq = Bio::Seq->new(-seq              => 'actgtggcgtcaact',
                     -description      => 'Sample Bio::Seq object',
                     -display_id       => 'something',
                     -accession_number => 'accnum',
                     -alphabet         => 'dna' );
```

Accessing remote databases (Bio::DB::GenBank, etc)

```
$gb = new Bio::DB::GenBank;

# this returns a Seq object :
$seq1 = $gb->get_Seq_by_id('MUSIGHBA1');

# this also returns a Seq object :
$seq2 = $gb->get_Seq_by_acc('AF303112');

# this returns a SeqIO object, which can be used to get a Seq object :
$seqio = $gb->get_Stream_by_id(["J00522","AF303112","2981014"]);
$seq3 = $seqio->next_seq;
```

Another common sequence manipulation task for nucleic acid sequences is locating restriction enzyme cutting sites. Bioperl provides the Bio::Restriction::Enzyme, Bio::Restriction::EnzymeCollection, Bio::Restriction::Analysis objects for this purpose. These modules replace the older module Bio::Tools::RestrictionEnzyme. A new collection of enzyme objects would be defined like this:

```
use Bio::Perl;
use Bio::Restriction::EnzymeCollection;

my $ae = Bio::Restriction::EnzymeCollection->new();

my $six_cutter_collection = $ae->cutters(6);

for my $enz ($six_cutter_collection)

{ print $enz->name,"\t",$enz->site,"\t",$enz->overhang_seq,"\n";

# prints name, recognition site, overhang
}
```

```
$seqobj = Bio::Seq->new(-seq => "AALLHHHHHHGGGGPPRTTTTTVVVVVVVVVVVVVV");

  use Bio::Tools::Sigcleave;

 $sigcleave_object = new Bio::Tools::Sigcleave ( -seq => $seqobj,
-threshold => 3.5, -description => 'test sigcleave protein seq', );

%raw_results = $sigcleave_object->signals;

$formatted_output = $sigcleave_object->pretty_print;
```

Running BLAST (using RemoteBlast.pm)

Bioperl supports remote execution of blasts at NCBI by means of the RemoteBlast object.

```
$remote_blast = Bio::Tools::Run::RemoteBlast->new (
         -prog => 'blastp', -data => 'ecoli', -expect => '1e-10' );
$r = $remote_blast->submit_blast("t/data/ecolitst.fa");
while ( @rids = $remote_blast->each_rid ) {
    for $rid ( @rids ) {
       $rc = $remote_blast->retrieve_blast($rid);
    }
}
```

```
$Bio::Tools::Run::RemoteBlast::HEADER{'MATRIX_NAME'} = 'BLOSUM25';
```

```
$input = Bio::Seq->new(-id  =>"test query",
                       -seq =>"ACTAAGTGGGGG");
```