File testing

Like BASH, file tests exist in Perl (source: `man perlfunc`):

```
-r  File is readable by effective uid/gid.
-w  File is writable by effective uid/gid.
-x  File is executable by effective uid/gid.
-o  File is owned by effective uid.

-R  File is readable by real uid/gid.
-W  File is writable by real uid/gid.
-X  File is executable by real uid/gid.
-O  File is owned by real uid.
```

-e	File exists.
-z	File has zero size (is empty).
-s	File has nonzero size (returns size in bytes).
-f	File is a plain file.
-d	File is a directory.
-l	File is a symbolic link.
-p	File is a named pipe (FIFO), or Filehandle is a pipe.
-S	File is a socket.
-b	File is a block special file.
-c	File is a character special file.
-t	Filehandle is opened to a tty.
-u	File has setuid bit set.
-g	File has setgid bit set.
-k	File has sticky bit set.

```
-T  File is an ASCII text file (heuristic guess).
-B  File is a "binary" file (opposite of -T).

-M  Script start time minus file modification time, in days.
-A  Same for access time.
-C  Same for inode change time (Unix, may differ for other platforms)
```

Using file status

You can use file status like this, for instance, as pre-test:

```
while (<>) {
    chomp;
    next unless -f $_;      # ignore specials
    #...
}
```

Using file status

Or you can use them as a post-test:

```
if(! open(FH, $fn))
{
  if(! -e "$fn")
  {
    die "File $fn doesn't exist.";
  }
  if(! -r "$fn")
  {
    die "File $fn isn't readable.";
  }
  if(-d "$fn")
  {
    die "$fn is a directory, not a regular file.";
  }
  die "$fn could not be opened.";
}
```

Subroutines in Perl

You can declare subroutines in Perl with sub, and call them with the & syntax:

```
my @list = qw( /etc/hosts /etc/resolv.conf /etc/init.d );
map ( &filecheck ,  @list) ;

sub filecheck
{
    if(-f "$_")
    {
        print "$_ is a regular file\n";
    }
    else
    {
        print "$_ is not a regular file\n";
    }
}
```

Subroutine arguments

To send arguments to a subroutine, just use a list after the subroutine invocation, just as you do with built-in functions in Perl.

Arguments are received in the `@_` array:

```
#!/usr/bin/perl -w
# 2006 10 04 - rdl Script39.pl
# shows subroutine argument lists
use strict;
my $val = max(10,20,30,40,11,99);
print "max = $val\n";
```

```
sub max
{
    print "Using $_[0] as first value...\n";
    my $memory = shift(@_);
    foreach(@_)
    {
        if($_ > $memory)
        {
            $memory = $_;
        }
    }
    return $memory;
}
```

Using `my` variables in subroutines

You can locally define variables for a subroutine with `my`:

```
sub func
{
  my $ct = @_;
  ...;
}
```

The variable `$ct` is defined only within the subroutine `func`.

sort() and map()

The built-ins functions `sort()` and `map()` can accept a subroutine rather than just an anonymous block:

```
@list = qw/ 1 100 11 10 /;
@default = sort(@list);
@mysort = sort {&mysort} @list;
print "default sort: @default\n";
print "mysort: @mysort\n";
sub mysort
{
  return $a <=> $b;
}
# yields
```

```
default sort: 1 10 100 11
mysort: 1 10 11 100
```

As you can see, `sort()` sends along two special, predefined variables, `$a` and `$b`.

`cmp` and friends

As discussed earlier, <=> returns a result of -1,0,1 if the left hand value is respectively numerically less than, equal to, or greater than the right hand value.

`cmp` returns the same, but uses lexical rather numerical ordering.

grep

A very similar operator is grep, which only returns a list of the items that matched an expression (sort and map should always return a list exactly as long as the input list.)

For example:

```
@out = grep {$_ % 2} qw/1 2 3 4 5 6 7 8 9 10/;
print "@out\n";
# yields
1 3 5 7 9
```

Notice that the block item should return 0 for non-matching items.

Directory operations

```
chdir $DIRNAME;          # change directory to $DIRNAME

glob $PATTERN;           # return a list of matching patterns
# example:
@list = glob "*.pl";
print "@list \n";
Script16.pl Script18.pl Script19.pl Script20.pl Script21.pl [...]
```

Manipulating files and directories

```
unlink $FN1, $FN2, ...;     # remove a hard or soft link to files

rename $FN1, $FN2;          # rename $FN1 to new name $FN2

mkdir $DN1;                 # create directory with umask default permissions

rmdir $DN1, $DN2, ...;      # remove directories

chmod perms, $FDN1;         # change permissions
```

Traversing a directory with opendir and readdir

You can pull in the contents of a directory with opendir and readdir:

```
opendir(DH,"/tmp");
@filenams = readdir(DH);
closedir(DH);
print "@filenams\n";
# yields
.s.PGSQL.5432.lock .. mapping-root ssh-WCWcZf4199 xses-langley.joHONt . OSL_PIPE_50
```

Calling other processes

In Perl, you have four convenient ways to call (sub)processes: the backtick function, the `system()` function, `fork()/exec()`, and `open()`.

The backtick function is the most convenient one for handling most output from subprocesses. For example

```
@lines = `head -10 /etc/hosts`;
print "@lines\n";
```

You can do this type of output very similarly with `open`, but `open` also allows you do conveniently send input to subprocesses.

`exec()` lets you change the present process to another executable; generally, this is done with a `fork()` to create a new child subprocess first.

The `system()` subroutine is a short-cut way of writing `fork`/`exec`. Handinding input and output, just as with `fork/exec` is not particularly convenient.