You can do anything with Perl

1. Write a Perl program, `tags.pl` which given the URL of a web page fetches it by running *wget* and prints the HTML tags it uses.
   The tag should be converted to lower case and printed in sorted order with a count of often each is used.

   Don't count closing tags.

   Make sure you don't print tags within HTML comments.

   For example:

```
$ ./tags.pl http://www.cse.unsw.edu.au
a 141
body 1
br 14
div 161
em 3
footer 1
form 1
h2 2
h4 3
h5 3
head 1
header 1
hr 3
html 1
img 12
input 5
li 99
link 3
meta 4
noscript 1
p 18
script 14
small 3
span 3
strong 4
title 1
ul 25
```

   Note the counts in the above example will not be current - the CSE pages changes almost daily.

   Sample solution for tags.pl

```
#!/usr/bin/perl -w
# written by andrewt@cse.unsw.edu.au as a COMP2041 example
# fetch specified web page and count the HTML tags in them

# There are better ways to fetch web pages (e.g. HTTP::Request::Common)
# The regex code below doesn't handle a number of cases.  It is often
# better to use a library to properly parse HTML before processing it.
# But beware illegal HTML is common & often causes problems for parsers.

foreach $url (@ARGV) {
    $webpage = `wget -q -O- '$url'`;
    $webpage =~ tr/A-Z/a-z/;
    $webpage =~ s/<!--.*?-->//g; # remove comments
    @tags = $webpage =~ /<\s*(\w+)/g;
    foreach $tag (@tags) {
        $tag_count{$tag}++;
    }
}
foreach $tag (sort keys %tag_count) {
    print "$tag $tag_count{$tag}\n";
}
```

2. Add an -f option to tags.pl which indicated the tags are to printed in order of frequency.

```
$ tags.pl -f http://www.cse.unsw.edu.au
head 1
noscript 1
html 1
form 1
title 1
footer 1
header 1
body 1
h2 2
hr 3
h4 3
span 3
link 3
small 3
h5 3
em 3
meta 4
strong 4
input 5
img 12
br 14
script 14
p 18
ul 25
li 99
a 141
div 161
```

Sample solution for tags.pl

```
#!/usr/bin/perl -w
# written by andrewt@cse.unsw.edu.au as a COMP2041 example
# fetch specified web page and count the HTML tags in them

# The regex code below doesn't handle a number of cases.  It is often
# better to use a library to properly parse HTML before processing it.
# But beware illegal HTML is common & often causes problems for parsers.

use LWP::Simple;

$sort_by_frequency = 0;
foreach $arg (@ARGV) {
    if ($arg eq "-f") {
        $sort_by_frequency = 1;
    } else {
        push @urls, $arg;
    }
}
foreach $url (@urls) {
    $webpage = get $url;
    $webpage =~ tr/A-Z/a-z/;
    $webpage =~ s/<!--.*?-->//g; # remove comments
    @tags = $webpage =~ /<\s*(\w+)/g;
    foreach $tag (@tags) {
        $tag_count{$tag}++;
    }
}
if ($sort_by_frequency) {
    @sorted_tags = sort {$tag_count{$a} <=> $tag_count{$b}} keys %tag_count;
} else {
    @sorted_tags = sort keys %tag_count;
}
print "$_ $tag_count{$_}\n" foreach @sorted_tags;
```

3. Write a Perl function which takes an integer argument n and reads the next n lines of input and returns them as a string.

Two sample solutions with extra code to run the function:

```
#!/usr/bin/perl -w

$n = shift @ARGV or die "Usage: $0 <n-lines>\n";

sub n_lines0 {
    my ($n) = @_;
    my $text = "";
    while ($n-- > 0) {
        $text .= <>;
    }
    return $text;
}

sub n_lines1 {
    my ($n) = @_;
    my $text = "";
    $text .= <> foreach (1..$n);
    return $text;
}

print n_lines1($n);
```

4. Write a Perl program `./shpl.pl` which reads Shell and outputs Perl for example:

```
while ls important_file >/dev/null
do
        echo "all OK"
        sleep 1
done
echo "Panic important_file gone"
```

Your program should produce this output:

```
while (!system "ls important_file >/dev/null") {
        print "all OK\n";
        system("sleep 1");
}
print "Panic important_file gone\n";
```

Assume that the only Shell features that you need to handle specially are while loops & echo statements, and assume the Shell is nicely formatted. Don't worry about the "#!" line.
Write a quick version using regexes which does no checking just translates

Then write a longer version that checks the syntax of the Shell while loops are correct, including handling nested while loops. Hint: use recursive functions.

Readable implementation of line-by-line translation with no checking.

```
#!/usr/bin/perl -w

use strict;

while (my $line = <>) {
    chomp $line;
    if ($line =~ /^(\s+)/) {
        print $1; # preserve indents
        $line =~ s/^\s+//;
    }
    $line =~ s/\s+$//;

    if ($line =~ /^#!/) {
        print "#!/usr/bin/perl -w\n";
    } elsif ($line =~ /while/) {
        $line =~ s/while //;
        print "while (!system \"$line\") {\n";
    } elsif ($line =~ /^do$/) {
        next;
    } elsif ($line =~ /done$/) {
        print "}\n";
    } elsif ($line =~ /echo/) {
        $line =~ s/echo //;
        $line =~ s/\"//g;
        print "print \"$line\\n\";\n";
    } elsif ($line =~ /sleep/) {
        print "system(\"$line\")\n";
    } elsif ($line =~ /^\s*#/ || $line =~ /^\s*/) {
        print "$line\n";
    } else { # Lines we can't translate are turned into comments
        print "#$line\n";
    }
}
```

More concise implementation of line-by-line translation with no checking. Note use of $_. Style of first solution preferable for its readability & maintainability, if you are thinking about the assignment

```
#!/usr/bin/perl -w

while (<>) {
    if (/^(\s*)while\s+(.*)/) {
        print "$1while (!system \"$2\") {\n";
    } elsif (/^\s*do\b/) {
    } elsif (/^(\s*)done\b/) {
        print "$1}\n";
    } elsif (/^(\s*)echo\s+"(.*)"/) {
        print "$1print \"$2\";\n";
    } elsif (/^(\s*)(\S.*)/) {
        print "$1system \"$2\";\n";
    } else {
        print;
    }
}
```

Even more concise but even less readable implementation of line-by-line translation with no checking. Note use of -p command line option. Style is useful for quick code for one-off use. Do not use this style for the assignment.

```
#!/usr/bin/perl -wp
s/^\s*do\b/{/ or
s/^(\s*while)\s+(.*)\n/$1 (!system \"$2\") / or
s/^(\s*)done\b/$1}/ or
s/^(\s*)echo\s+"(.*)"/$1print \"$2\\n\";/ or
s/^(\s*)(\S.*)/$1system \"$2\";/;
```

Sample recursive implementation which checks while loop syntax.

```
#!/usr/bin/perl -w

sub translate_statement {
        if ($shell[0] =~ /\s*while\b/) {
                return translate_while();
        } elsif ($shell[0] =~ /\s*echo\b/) {
                return translate_echo();
        } else{
                my $shell = shift @shell;
                if ($shell =~ /^(\s*)(\S.*)/) {
                        return "$1system \"$2\";\n"
                } else {
                        return $shell;
                }
        }
}

sub translate_while {
        my @while_perl = ();
        my $shell = shift @shell;
        $shell =~ /^(\s*)while\s+(.*)/ or die;
        push @while_perl, "$1while (!system \"$2\") {\n";
        $shell = shift @shell;
        die "syntax error: expected do" if $shell !~ /^\s*do\b/;
        while (@shell && $shell[0] !~ /^(\s*)done\b/) {
                push @while_perl, translate_statement();
        }
        $shell = shift(@shell) || "";
        $shell =~ /^(\s*)done\b/ or die "syntax error: expected done";
        push @while_perl, "$1}\n";
        return @while_perl;
}

sub translate_echo {
        my $shell = shift @shell;
        $shell =~ /^(\s*)echo\s+"(.*)"/ or die;
        return "$1print \"$2\\n\";\n";
}

# global array containing input shell statements is
# is consumed as Shell is translated to Perl

@shell = <>;
@perl = ();
while (@shell) {
        push @perl, translate_statement();
}
print @perl;
```

Sample recursive implementation which checks while loop syntax and does its own indentation rather hten relying on the shell script being correctly indented.

```
#!/usr/bin/perl -w

sub translate_statement {
        my ($indent) = @_;
        if ($shell[0] =~ /\s*while\b/) {
                return translate_while($indent);
        } elsif ($shell[0] =~ /\s*echo\b/) {
                return translate_echo($indent);
        } else {
                my $shell = shift @shell;
                if ($shell =~ /^\s*(\S.*)/) {
                        return (" " x $indent)."system \"$1\";\n"
                } else {
                        return "\n";
                }
        }
}

sub translate_while {
        my ($indent) = @_;
        my @while_perl = ();
        my $shell = shift @shell;
        $shell =~ /^\s*while\s+(.*)/ or die;
        push @while_perl, (" " x $indent)."while (!system \"$1\") {\n";
        $shell = shift @shell;
        die "syntax error: expected do" if $shell !~ /^\s*do\b/;
        while (@shell && $shell[0] !~ /^\s*done\b/) {
                push @while_perl, translate_statement($indent + 4);
        }
        $shell = shift(@shell) || "";
        $shell =~ /^\s*done\b/ or die "syntax error: expected done";
        push @while_perl, (" " x $indent)."}\n";
        return @while_perl;
}

sub translate_echo {
        my ($indent) = @_;
        my $shell = shift @shell;
        $shell =~ /^\s*echo\s+"(.*)"/ or die;
        return  (" " x $indent)."print \"$1\\n\";\n";
}

# global array containing input shell statements is
# is consumed as Shell is translated to Perl

@shell = <>;
@perl = ();
while (@shell) {
        push @perl, translate_statement(0);
}
print @perl;
```

5. Give Perl code which given the name of a C function searches the C source files (*.c) in the current directory for calls of the function, declarations & definitons of the function and prints a message indicating the file and line number, in the format below.
   You can assume functions are defined with the type, name and paramaters on a single non-indented line. You can assume function bodies are always indented.

   You don't have to handle multi line comments. Try to avoid matching the function name in strings or single line comments
   For example:

```
$ cat half.c
double half(double x) {
    return  x/2;
}
$ cat main.c
#include <stdio.h>
#include <stdlib.h>

double half(double x);

int main(int argc, char *argv[]) {
    return  half(atoi(argv[1]));
}
$ ./print_function_uses.pl half
a.c:1 function half defined
half.c:1 function half defined
main.c:4 function half declared
main.c:7 function half used
```

Perl sample solution

```
#!/usr/bin/perl -w

$function = $ARGV[0] or die "Usage: $0 <function-name>\n";

foreach $c_file (glob "*.c") {
    open my $cf, '<', $c_file or die "$0: can not open $c_file: $!\n";
    while ($line = <$cf>) {
        # remove single-line comments & strings (breaks if strings contain ")
        $line =~ s/\/\/.*//;
        $line =~ s/\/\*.*?\*\///;
        $line =~ s/".*?"//;
        # note use of \b (word boundary) to match function
        $line =~ /\b$function\s*\(/ or next;
        print "$c_file:$. function $function ";
        # if line is indented it should be a call to the function
        if ($line =~ /^\s/) {
            print "used\n";
        } elsif ($line =~ /;/) {
            print "declared\n";
        } else {
            print "defined\n";
        }
    }
    close $cf;
}
```

Python sample solution

```
#!/usr/bin/python
import glob, sys, re

if len(sys.argv) != 2:
    sys.stdout.write("Usage: %s <function-name>\n\n" % sys.argv[0])
    sys.exit(1)

function = sys.argv[1]

for c_file in glob.glob("*.c"):
    with open(c_file) as cf:
        # note use of \b (word boundary) to match function
        function_regex = r'\b%s\s*\(' % function
        line_number = 0
        for line in cf:
            line_number = line_number + 1
            # remove single-line comments & strings (breaks if strings contain ")
            line = re.sub(r'\/\/.*', '', line)
            line = re.sub(r'".*?"', '', line)
            if not re.search(function_regex, line):
                continue
            # if line is indented it should be a call to the function
            if re.search(r'^\s', line):
                which = "used"
            elif re.search(r';', line):
                which = "declared"
            else:
                which = "defined"
            print("%s:%d function %s %s" % (c_file, line_number, function,  which))
```

6. Give Perl code which given a C program as input finds the definitions of single parameter functions and prints separately the function's type, name and the parameters name & type. Assume all these occur on a single non-indented line in the C source code. You can assume function bodies are always indented. Allow for white space occuring anywhere in the function header. You can assume that types in the program don't contain square or round brackets. For example:

```
$ cat a.c
double half(int *x) {
    return  *x/2.0;
}
$ ./print_function_types.pl a.c
function type='double'
function name='half'
parameter type='int *'
parameter name='x'
```

Perl sample solution

```
#!/usr/bin/perl -w

while ($line = <>) {
    $line =~ /^([a-zA-Z_].*)\((.*)\)/ or next;
    $function_start = $1;
    $parameter = $2;
    $function_type = $function_start;
    $function_type =~ s/\s*([a-zA-Z_]\w*)\s*$// or next;
    $function_name = $1;
    $parameter_type = $parameter;
    $parameter_type =~ s/\s*([a-zA-Z_]\w*)\s*$// or next;
    $parameter_name = $1;
    print "function type='$function_type'\n";
    print "function name='$function_name'\n";
    print "parameter type='$parameter_type'\n";
    print "parameter name='$parameter_name'\n";
}
```

7. Write a Perl script C_include.pl which given the name of a C source file prints the file replacing any '#include' lines with the contents of the included file, if the included file itself contains a #include line these should also be processed.
   Assume the source files contain only quoted ("") include directives which contain the files's actual path name. For example:

```
$ cat f.c
#include "true.h"

int main(int argc, char *argv[]) {
    return  TRUE || FALSE;
}
$ cat true.h
#define TRUE 1
#include "false.h"
$ cat false.h
#define FALSE 0
$ ./C_include.pl f.c
#define TRUE 1
#define FALSE 0

int main(int argc, char *argv[]) {
    return  TRUE || FALSE;
}
```

Perl sample solution

```
#!/usr/bin/perl -w
# Given C source files interpolate #include "FILE" directives recursively.
sub include_file($);

sub include_file($) {
    my ($file) = @_;
    # this function is recursive so a local filehandle is essential
    open my $f, '<', $file or die "$0: can not open $file: $!";
    while ($line = <$f>) {
        if ($line =~ /^#\s*include\s*"([^"]*)"/) {
            include_file($1);
        } else {
            print $line;
        }
    }
    close $f;
}

foreach $file (@ARGV) {
    include_file($file);
}
```

8. Modify C_include.pl so that it handles both "" and <> directives. It should search the directories `/usr/include/`, `/usr/local/include` and `/usr/include/x86_64-linux-gnu` for include files specified in <> directives and for files specified in "" directives which do not exist locally. For example:

```
$ cat g.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("hello world\n");
    exit(0);
}
$ ./C_include.pl g.c
/* Define ISO C stdio on top of C++ iostreams.
   Copyright (C) 1991, 1994-2008, 2009, 2010 Free Software Foundatio
   This file is part of the GNU C Library.

   The GNU C Library is free software; you can redistribute it and/o
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version

   The GNU C Library is distributed in the hope that it will be usef
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with the GNU C Library; if not, write to the Free
   Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
   02111-1307 USA.  */

/*
 *      ISO C99 Standard: 7.19 Input/output     <stdio.h>
 */

#ifndef _STDIO_H

#if !defined __need_FILE && !defined __need___FILE
# define _STDIO_H       1
/* Copyright (C) 1991-1993,1995-2007,2009,2010,2011
   Free Software Foundation, Inc.
   This file is part of the GNU C Library.

   The GNU C Library is free software; you can redistribute it and/o
...
```

Perl sample solution

```
#!/usr/bin/perl -w
# Given C source files interpolate #include "FILE" and #include <FILE>
# directives recursively.
# The recursion in this script not terminate on stdio.h etc
# because #ifdef directive are not handled

@include_dirs = ('/usr/include/','/usr/local/include','/usr/include/x86_64-linux-gnu');

sub include_file($@);

sub include_file($@) {
    my ($file,@prefixes) = @_;
    foreach $prefix (@prefixes) {
        # this function is recursive so a local filehandle is essential
        my $path = "$prefix$file";
        next if !-r $path;
        open my $f, '<', $path or die "$0: can not open $path: $!";
        while ($line = <$f>) {
            if ($line =~ /^#\s*include\s*"(.*?)"/) {
                include_file($1, ('',@include_dirs));
            } elsif ($line =~ /^#\s*include\s*<(.*?)>/) {
                include_file($1, @include_dirs);
            } else {
                print $line;
            }
        }
        close $f;
        return;
    }
    die "$0: can not find: $file\n";
}

foreach $file (@ARGV) {
    include_file($file, (''));
}
```

9. Write a Perl program, `times.pl` which prints a table of multiplications.
   Your program will be given the dimension of the table and the width of the columns to be printed. For example:

```
$ times.pl 4 5 3
  1   1   2   3   4   5
  2   2   4   6   8  10
  3   3   6   9  12  15
  4   4   8  12  16  20
```

Sample Perl solution

```
#!/usr/bin/perl -w
die "Usage $0 <n> <m> <column-width>" if @ARGV != 3;
$n = $ARGV[0];
$m = $ARGV[1];
$width = $ARGV[2];
$format = "%${width}d";
foreach $x (1..$n) {
    printf $format, $x;
    foreach $y (1..$m) {
        printf "%${width}d", $x*$y;
    }
    print "\n";
}
```

Sample Python solution

```
#!/usr/bin/python
import glob, sys, re

if len(sys.argv) != 4:
    sys.stdout.write("Usage: %s <n> <m> <column-width>\n\n" % sys.argv[0])
    sys.exit(1)

n = int(sys.argv[1])
m = int(sys.argv[2])
width = int(sys.argv[3])
format = "%%%dd" % width

for x in range(1, n + 1):
    print(format % x)
    for y in range(1, m + 1):
        sys.stdout.write(' ' + (format % (x * y)))
    print()
```

10. Write a Perl program which deletes blank lines from each of the files specified as arguments. For example, if run like this:

```
$ deblank.pl file1 file2 file3
```

your program should delete any blank lines in `file1`, `file2` and `file3`. Note that this program *changes* the files, it doesn't just write the "de-blanked" versions to standard output.

Perl sample solution

```
#!/usr/bin/perl -w
# delete blank lines from specified files

die "Usage: $0 <files>\n" if !@ARGV;

foreach $file (@ARGV) {
    open my $in, '<', $file or die "Can not open $file: $!";
    @lines = <$in>;  # reads entire file into array
    close $in;
    open my $out, '>', $file or die "Can not open $file: $!";
    foreach $line (@lines) {
        print $out $line if $line !~ /^\s*$/;
    }
    close $out;
}
```

Perl sample solution using -i switch

```
#!/usr/bin/perl -w -i
while (<>) {
    print if !/^\s*$/;
}
```

Perl sample solution using -i and -p switch

```
#!/usr/bin/perl -w -i -p
s/^\s*$//
```

Or from the command line:

```
$ perl -ip -e 's/^\s*$//' file1 file2 file3
```

Python sample solution - based on Perl

```
#!/usr/bin/python
# delete blank lines from specified files
# simple code which could lose data, if a write error occurs
import sys, re

for filename in sys.argv[1:]:
    with open(filename) as f:
        lines = f.readlines()
    with open(filename,'w') as f:
        for line in lines:
            if not re.match(r'^\s*$', line):
                f.write(line)
```

11. Write a Perl function `listToHTML()` that given a list of values returns a string of HTML code as an unordered list:

```
out = listToHTML('The', 'Quick', 'Brown', 'Fox');
```

would result in `$out` having the value ...

```
<ul>
<li>The
<li>Quick
<li>Brown
<li>Fox
</ul>
```

As part of an HTML page, this would display as:

- The
- Quick
- Brown
- Fox

P.S. A Perl syntactic short cut can be used to construct the list above:

```
out = listToHTML(qw/The Quick Brown Fox/);
```

Sample solution for listToHTML

```
#!/usr/bin/perl -w

sub listToHTML(@) {
    my (@list) = @_;
    return "" if !@list;
    return "<ul>\n<li>".join("\n<li>", @list)."\n</ul>\n";
}
print listToHTML(@ARGV);
```

12. Write a Perl function `hashToHTML()` that returns a string of HTML code that could be used to display a Perl associative array (hash) as an HTML table, e.g.

```
# the hash table ...
colours = ("John"=>"blue", "Anne"=>"red", "Andrew"=>"green");
# and the function call ...
out = hashToHTML(%colours);
```

would result in `$out` having the value ...

```
<table border="1" cellpadding="5">
<tr><th> Key </th><th> Value </th></tr>
<tr><td> Andrew </td><td> green </td></tr>
<tr><td> Anne </td><td> red </td></tr>
<tr><td> John </td><td> blue </td></tr>
</table>
```

As part of an HTML page, this would display as:

Key	**Value**
Andrew	green
Anne	red
John	blue

Note that the hash should be displayed in ascending alphabetical order on key values.

This gives the function as well as some code to test it out:

```
#!/usr/bin/perl -w

sub hashToHTML {
    my (%tab) = @_;
    my $html = "";

    $html = "<table border=\"1\" cellpadding=\"5\">\n".
        "<tr><th> Key </th><th> Value </th></tr>\n";

    foreach $k (sort keys %tab) {
        $html .= "<tr><td> $k </td><td> $tab{$k} </td></tr>\n";
    }
    $html .= "</table>\n";
    return $html;
}

%h = ("David"=>"green", "Phil"=>"blue", "Andrew"=>"red", "John"=>"blue");

print hashToHTML(%h);
exit;
```

13. Write a perl program that will read in an HTML document and output a new HTML document that contains a table with two cells (in one row). In the left cell should be a copy of the complete original HTML document inside <PRE> tags so we can see the raw HTML. You will need to replace all "<" characters with the sequence "<" and all ">" characters with the sequence ">", otherwise the browser will think they are HTML tags (and we want to see the tags in the left cell). In the right cell just include the HTML body of the document, so we can see what it will look like when rendered by a browser.

Sample solution for show_html.pl

```
#!/usr/bin/perl -w
# inspired by from www.cs.www.cs.rpi.edu/~hollingd/eiw.old/5-Perl/ex6.html

my $html_source = join "", <>;
my $modified_html = $html_source;
$modified_html =~ s/<\s*HEAD[^>]*>.*?<\s*\/HEAD[^>]*>//si;
$modified_html =~ s/<\s*\/?\s*(BODY|HTML)[^>]*>//gsi;

my ($title) = ($html_source =~ /.*<\s*TITLE[^>]*>(.*?)<\s*\/TITLE[^>]*>/si);
$title = "No title" if !defined $title;

$html_source =~ s/</\&lt;/g;
$html_source =~ s/>/\&gt;/g;

print <<eof;
<HTML>
<HEAD>
<TITLE>$title</TITLE>
</HEAD>
<BODY>
<H3 ALIGN=CENTER>HTML-VIEW of $title</H3>
<TABLE BORDER=1 BGCOLOR=WHEAT>
<TR><TD><PRE><FONT SIZE=SMALL>$html_source</FONT></PRE></TD><TD>$modified_html</TD></TR>
</TABLE>
</BODY>
</HTML>
eof
```

14. Write a Perl program that reads in data about student performance in a Prac Exam and computes the overall result for each student. The program takes a *single command line argument*, which is the name of a file containing space-separated text records of the form:

```
studentID  exerciseID  testsPassed  numWarnings
```

There will be one line in the file for each exercise submitted by a student, so a given student may have one, two or three lines of data.
The output is ordered by student ID and contains a single line for each student, in the format:

```
studentID  totalMark  passOrFail
```

The *totalMark* value is computed as follows:

- if an exercise passes all 5 tests, it is awarded a mark of 10 and is *correct*
- if an exercise passes less than 5 tests, it is awarded a mark of *testsPassed/2* and is *incorrect*
- if there are *any* warnings on an exercise, the mark is reduced by 2
- the minimum mark for a given exercise is zero
- the *totalMark* is the sum of the marks for the individual exercises

The *totalMark* value should be display using the `printf` format `"%4.1f"` . A student is awarded a `PASS` if they have 2 or 3 *correct* exercises and is awarded a `FAIL` otherwise. Note that warnings do not cause an exercise to be treated as incorrect.

Sample Marks File	**Corresponding Output**
Command line argument: `marks1`	

`2121211 ex1 5 0`
`2121211 ex2 5 0`
`2121211 ex3 5 0`
`2233455 ex1 5 0`
`2233455 ex2 5 1`
`2233455 ex3 0 1`
`2277688 ex1 4 0`
`2277688 ex2 3 0`
`2277688 ex3 2 1`
`2277689 ex1 5 0`
`2277689 ex2 5 0`
`2277689 ex3 1 1`
	`2121211 30.0 PASS`
	`2233455 18.0 PASS`
	`2277688  3.5 FAIL`
	`2277689 20.0 PASS`

Sample Perl solution

```
#!/usr/local/bin/perl
#
# Prac Exam Exercise
# Author: John Shepherd (sample solution)
#

while (<>) {
    chomp;
    my ($sid,$ex,$tests,$warns) = split;
    if ($tests == 5) {
        $mark = 10;
        $ncorrect{"$sid"}++;
    }
    else {
        $mark = $tests/2.0;
    }
    $mark -= 2 if ($warns > 0);
    $mark = 0 if ($mark < 0);
    $total{$sid} += $mark;
}

foreach $sid (sort keys %total) {
    if ($ncorrect{$sid} >= 2) {
        $passfail = "PASS";
    } else {
        $passfail = "FAIL";
    }
    printf "%s %4.1f %s\n", $sid, $total{$sid}, $passfail;
}
```

Sample Python solution

```
#!/usr/bin/python
import fileinput, re, sys, collections
ncorrect = collections.defaultdict(int)
total = collections.defaultdict(float)
for line in fileinput.input():
    (sid,ex,tests,warns) = line.split()
    if tests == '5':
        mark = 10
        ncorrect[sid] += 1
    else:
        mark = int(tests)/2.0
    if int(warns) > 0:
        mark = max(0, mark - 2)
    total[sid] += mark
for sid in sorted(total.keys()):
    if ncorrect[sid] >= 2:
        passfail = "PASS"
    else:
        passfail = "FAIL"
    print("%s %4.1f %s" % (sid, total[sid], passfail))
```

15. What does this Perl print and why?

```
@a = (1..5);
@b = grep { $_ = $_ - 3; $_ > 0} @a;
print "@a\n";
print "@b\n";
```

It prints:

```
-2 -1 0 1 2
1 2
```

The `grep` function aliases $_ to each list element in turn an executes the code in the block. It returns a list ofthe element for which the last expressione evaluated is true.

`{ $_ = $_ - 3 }` subtracts 3 from each element in `@a` . The `$_ > 0` expression selects positive elements.

16. What does this Perl print?

```
@vec = map { $_ ** 2 } (1,2,3,4,5);
print "@vec\n";
```

It prints:

```
1 4 9 16 25
```

The `map` function applies the code in the block `{ $_ ** 2 }` to each element in the list, and returns a list containing the tranformed values. The `**` operator does exponentiation; and `$_` refers to the "current" element in the list.