Week 09 Tutorial Sample Answers

1. Discuss how Perl can be generated for the supplied examples for subsets 0-3

   Answer:

   Discussed in tut.

2. Discuss the assignment specification and possible strategies for the assignment.

   Answer:

   Discussed in tut.

3. Suppose typing `ls -l` yields:

```
-rw-r--r--    1 cs2041   cs2041         99 Sep 14 10:14 a
-rw-r--r--    1 cs2041   cs2041         26 Oct 20 18:16 b
-rw-r--r--    1 cs2041   cs2041         13 Sep 14 10:14 Makefile
```

   and typing `more Makefile` yields:

```
a: b
	cp b a
```

   What happens if `make` is typed?

   What happens if `make` is typed a second time?

   Answer:

   `cp b a`

   Nothing.

4. The following is an attempt by an inexperienced developer to produce a Makefile for a small project consisting of a main program (`main.c`), one module (`module.c` and `module.h`) plus a file of common definitions (`defs.h`). Both C files #include the two header files. The final product is called "myapp".

```
$CC=gcc-4.4
myapp:  main.o module.o defs.h
$CC -c -o $< main.o module.o

main.o:  module.h module.c defs.h
$CC -c main.o

module.o:  module.h defs.h
$CC -c main.o

main.c:  defs.h
```

   There are 6+ errors in the Makefile. Identify them and rewrite the Makefile so it correctly reflects the dependencies and rules for building `myapp`.

   Answer:

   - Incorrect variable syntax - the syntax is wrong for the assignment to the variable CC and in all of its uses.
   - Incorrect rule syntax - the build (compile) commands should be indented with a tab.
   - Rule 1: the final application only depends on the object files. It may indirectly depend on headers, but only through the object files.
   - Rule 1 command: the flag `-c` is used only for compiling, not for linking.
   - Rule 1 command: the implicit variable for the target is `$@`, not `$<`.
   - Rule 2: main.o depends on the header files and its own C file, not on any other C file (you don't have to recompile main.c if module.c changes).
   - Rule 3: module.o depends on module.c too.
   - Rule 4 doesn't make sense: main.c is a source file that doesn't depend on anything. Remove the rule.

   The revised Makefile is

```
CC=gcc-4.4

myapp: main.o module.o
    $(CC) -o $@ main.o module.o

main.o: main.c module.h defs.h
    $(CC) -c main.c

module.o: module.c module.h defs.h
    $(CC) -c module.c
```

5. Write a Perl script, `functions.pl` which when run in the same diectory as a multi-file C program, for every function, prints its prototype, where the file and line it is defined and the file and line for every, for example:

   For example:

   ```
   $ ./functions.pl
   function: get_world
     prototype: void get_world(FILE *f, int n)
     defined: world.c:3
     called: game.c:15 games.c:29 save.c:67
   function: main
     prototype: int main(void)
     defined: main.c:15
     called:
   .....
   ```

   Discuss what assumptions you have to make about C code.

   What C features might be confused with function calls.

   Answer:

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

# read all .c files in current directory
# and collections location of fucntion definitions and calls
#
# Brittle solution which makes assumptions which may not apply to real code
# For example assumes:
# 1) function prototype appears on a single
# 2) a function call is not spread over multiple lines
# 3) multiple function calls do not occur on a single
#
# code like this can still be useful to gather e.g. 90% of data

sub main {
    foreach $file (glob "*.c") {
        process_file($file);
    }

    foreach $function_name (sort keys %definition) {
        print "function: $function_name\n";
        print "  prototype: $prototype{$function_name}\n";
        print "  defined: $definition{$function_name}\n";
        print "  called:$calls{$function_name}\n" if $calls{$function_name};
    }
}


sub process_file {
    my ($file) = @_;
    our (%prototype, %definition, %calls);
    open F, $file or die "Can not open $file: $!";
    my $line;
    while ($line = <F>) {
        if ($line =~ /^([a-zA-Z_].*\b([a-zA-Z_][a-zA-Z0-9_]+)\(.*\))/) {
            my $proto = $1;
            my $function_name = $2;
            $prototype{$function_name} = $proto;
            $definition{$function_name} = "$file:$.";
        } elsif ($line =~ /\b([a-zA-Z_]\w+)\(.*\)/) {
            my $function_name = $1;
            $calls{$function_name} .= " $file:$.";
        }
    }
    close F;
}

main();
```

6. Write a Perl program that given the road distances between a number of towns (on standard input) calculates the shortest journey between two towns specified as arguments. Here is an example of how your program should behave.

```
$ ./shortest_path.pl Parkes Gilgandra
Bourke Broken-Hill   217
Bourke Dubbo          23
Bourke Gilgandra      62
Bourke Parkes         71
Canowindra Dubbo      35
Canowindra Gilgandra  13
Canowindra Parkes    112
Dubbo Gilgandra       91
Dubbo Parkes          57
Ctrl-D
Shortest route is length = 105: Parkes Dubbo Canowindra Gilgandra.
```

Answer:

Fairly obvious Perl sample solution

```perl
#!/usr/bin/perl -w
# find shortest path between two towns

die "Usage: $0 <start> <finish>\n" if @ARGV != 2;
$start = $ARGV[0];
$finish = $ARGV[1];

while (<STDIN>) {
    /(\S+)\s+(\S+)\s+(\d+)/ || next;
    $distance{$1}{$2} = $3;
    $distance{$2}{$1} = $3;
}

$shortest_journey{$start} = 0;
$route{$start} = "";
@unprocessed_towns = keys %distance;
$current_town = $start;
while ($current_town && $current_town ne $finish) {
    @unprocessed_towns = grep {$_ ne $current_town} @unprocessed_towns;

    foreach  $town (@unprocessed_towns) {
        if (defined $distance{$current_town}{$town}) {
            my $d = $shortest_journey{$current_town} + $distance{$current_town}{$town};
            if (!defined $shortest_journey{$town} || $shortest_journey{$town} > $d) {
                $shortest_journey{$town} = $d;
                $route{$town} = "$route{$current_town} $current_town";
            }
        }
    }

    my $min_distance = 1e99;   # must be larger than any possible distance
    $current_town = "";
    foreach $town (@unprocessed_towns) {
        if (defined $shortest_journey{$town} && $shortest_journey{$town} < $min_distance) {
            $min_distance = $shortest_journey{$town};
            $current_town = $town;
        }
    }
}

if (!defined $shortest_journey{$finish}) {
    print "No route from $start to $finish.\n";
} else {
    print "Shortest route is length = $shortest_journey{$finish}:$route{$finish} $finish.\n";
}
```

More concise Perl solution

```perl
#!/usr/bin/perl -w
# find shortest path between two towns

die "Usage: $0 <start> <finish>\n" if @ARGV != 2;
$start = $ARGV[0];
$finish = $ARGV[1];

while (<STDIN>) {
    /(\S+)\s+(\S+)\s+(\d+)/ || next;
    $distance{$1}{$2} = $3;
    $distance{$2}{$1} = $3;
}

$shortest_journey{$start} = 0;
$route{$start} = "";
$current_town = $start;
while ($current_town && $current_town ne $finish) {
    foreach  $town (keys %{$distance{$current_town}}) {
        my $d = $shortest_journey{$current_town} + $distance{$current_town}{$town};
        next if defined $shortest_journey{$town} && $shortest_journey{$town} < $d;
        $shortest_journey{$town} = $d;
        $route{$town} = "$route{$current_town} $current_town";
    }
    delete $distance{$current_town};
    my $min_distance = 1e99;   # must be larger than any possible distance
    $current_town = "";
    foreach $town (keys %distance) {
        next if !defined $shortest_journey{$town};
        next if $shortest_journey{$town} > $min_distance;
        $min_distance = $shortest_journey{$town};
        $current_town = $town;
    }
}

if (!defined $shortest_journey{$finish}) {
    print "No route from $start to $finish.\n";
} else {
    print "Shortest route is length = $shortest_journey{$finish}:$route{$finish} $finish.\n";
}
```