KIT213 UNIX Scripting Assignment

Due Date: 3PM Wednesday Week 13, 12 October 2022
Weight: 30% of your final KIT213 result

1. Introduction

This assignment requires you to apply what you have learned from the unit's practical classes to write a script to implement task specified below. You must complete this assignment individually and pay careful attention to the section on plagiarism **to ensure the work you submit is your own**.

2. Working environment

On the teaching server, `ictteach.its.utas.edu.au`, you must make a directory named `kit213script` in your home directory and use this directory as your *working directory* for all assignment development.

1. The first time you start working on the assignment, after logging on, type the following commands ($ is the prompt, do not type that):

```
$ mkdir kit213script
$ cd kit213script
```

2. Every other time you log on, simply do the following and then continue working:

```
$ cd kit213script
```

3. Allowed Syntax/Commands

All scripts must be completed using syntax and commands discussed in past (and some future) practical classes – no additional referential syntax (that is not discussed in practicals) is allowed. This means your script solutions must not use *alternative* constructs and syntax e.g. solutions found through websites via google searches that use commands and shell syntax that we have not covered in the practical content, and you must not use external parties or other individuals to write the scripts for you (this is considered plagiarism and academic misconduct).

Your solution is restricted (i.e **not allowed**) to use the shell "builtin" binary conversion syntax e.g. **`echo $((2#101010101))`**

This restriction also extends to any other command that has not been discussed in classes – the following list shows *some* of the common methods (**but not all**) found through a brief google search – **none are permitted**:

```
awk sed printf xxd od perl ibase, obase , bc
```

The unit also has not discussed *arrays*, so syntax similar in form to

```
powers=(128 64 32 16 8 4 2 1)   #an array
for i in ${!powers[@]}; do
   something ${powers[$i]}
done
```

is strictly prohibited.

Instead, your solution must take an iterative (looping) approach to demonstrate your understanding of iteration and selecting individual characters from a string (a string is just a sequence of characters). If in doubt as to what syntax or commands may be used, ask your tutor or the lecturer.

4. Assessment Process

All student script submissions will be marked on **ictteach** according to the rubric on the last page of this document. If your scripts were developed elsewhere (i.e., not on **ictteach**) it's your responsibility to ensure that your scripts run correctly in our teaching environment. You will lose considerable marks if your scripts do not run correctly on our teaching server, so it is imperative you develop and test your scripts on **ictteach** before you submit.

- Pay close attention to the marking rubric shown on the last page. This scheme shows you how marks are allocated, and what you must (successfully) achieve to receive marks. Note also that your scripts must have clear and tidy structure, have good use of indentation and whitespace (to make the script more readable by humans), include appropriate comments, and have your identification details and the script's purpose specified in comments at the top of your script.

5. Scripting Task Overview

5.1 Scenario

There is a long tradition of producing line-based character art for a terminal using the limited character representation available in the ASCII character set. In this assessment task, you are working for a fictional company that wants to create some simple command-line “banners” that include letters, numbers and possibly patterns from pre-defined binary data that will need to be converted from decimal format. The pre-defined decimal-format data for the alphabet and numerals has been created by the company’s art department.

Each individual pattern that will ultimately be shown in the terminal window will be based on 8 rows of binary data (an 8 by 8 grid of "pixels"). As an example, consider a representation of the letter 'A':

Column value								
128 2^7	64 2^6	32 2^5	16 2^4	8 2^3	4 2^2	2 2^1	1 2^0	Row Value
								0
		■	■	■				56
	■	■		■	■			108
■	■				■	■		198
■	■				■	■		198
■	■	■	■	■	■	■		254
■	■				■	■		198
■	■				■	■		198

Each column in the diagram above represents a power of two, and each entire row represents an 8-bit value (a byte), so the whole letter 'A' above uses 8 bytes for this representation. If a pixel (a particular row and column location) is white, that is the equivalent of a zero in the column for that row. If a pixel is black, it is the equivalent of a 1. In pure binary, the whole letter above is represented by the following binary patterns:

```
00000000
00111000
01101100
11000110
11000110
11111110
11000110
11000110
```

The value for the top row here is 0 in decimal. The next row has a 1 in the 2^5 column (32), a 1 in the 2^4 column (16), and a 1 in the 2^3 (8) column. This makes the second row (byte) value 32+16+8 = 56, the row value for the third row 64+32+8+4 = 108 etc.

5.2 Provided source "artwork" files

The entire uppercase alphabetic characters and the numeric characters 0-9 have been pre-drawn by the art department in a similar manner to that describe above. However, the data for these characters has unfortunately been generated as a **series of individual files** (with 8 files per character/pattern) that contain no data – instead, the encoded data binary value is part of each file's name as a decimal number.

Continuing with the letter 'A' example as well as the representation for the letters 'B' and 'C' (shown below), they are represented in the provided source files using the following filenames:

Letter 'A'	Letter 'B'	Letter 'C'
`AA1-0.grf`	`AB1-0.grf`	`AC1-0.grf`
`AA2-56.grf`	`AB2-252.grf`	`AC2-60.grf`
`AA3-108.grf`	`AB3-198.grf`	`AC3-102.grf`
`AA4-198.grf`	`AB4-198.grf`	`AC4-192.grf`
`AA5-198.grf`	`AB5-252.grf`	`AC5-192.grf`
`AA6-254.grf`	`AB6-198.grf`	`AC6-192.grf`
`AA7-198.grf`	`AB7-198.grf`	`AC7-102.grf`
`AA8-198.grf`	`AB8-252.grf`	`AC8-60.grf`

Column								
128 2^7	64 2^6	32 2^5	16 2^4	8 2^3	4 2^2	2 2^1	1 2^0	Row
								0
■	■	■	■	■	■			252
■	■				■	■		198
■	■				■	■		198
■	■	■	■	■	■			252
■	■				■	■		198
■	■				■	■		198
■	■	■	■	■	■			252

Column								
128 2^7	64 2^6	32 2^5	16 2^4	8 2^3	4 2^2	2 2^1	1 2^0	Row
								0
		■	■	■	■			60
	■	■			■	■		102
■	■							192
■	■							192
■	■							192
	■	■			■	■		102
		■	■	■	■			60

5.3 Source artwork original naming convention

The format of each provided source filename is the following, which is only important if you manually copy any of the files to your own local directory:

First character	Second character	Third character	Fourth character	Fifth (and possibly sixth and seventh) characters	Last 4 characters
A literal 'A'	The letter being represented (i.e. 'A' , 'B' , 'C' … 'Z', '0' .. '9')	The row number from each 8x8 grid, starting from 1	A dash (-)	The decimal value of the original byte for each row in the character's representation grid – this ranges from 0 .. 255	A file extension – the source files use `.grf`

All provided source art files can be found in the following directory:

```
/units/kit213/assignment
```

Try to list the contents of that directory (with one filename per line using the `-1` (*minus one*) option, (the `$` is the prompt)):

```
$ ls -1 /units/kit213/assignment
```

You will see in the output that the files are listed in a particular order – this is the reason for the filename convention used, as `ls` will list the files in alphabetic and numeric order, preserving the grouping of 8 files per represented character. There are some extra "symbols" that are also included at the end that you can also use/test with if you want to – these include:

- The space character (ZA…)
- A black solid square (ZB…)
- A square shape (ZC…)
- A triangle shape (ZD…)
- A checkerboard (ZE…)
- A diamond shape (ZF…)
- Diagonal left lines (ZG…)
- Diagonal right lines (ZH…)

5.4 Copying source artwork

Part of your assignment task will require you to initially copy some of the source artwork files to your own local directory for processing by the script you will be creating later, however, to simplify and aid with this copying, a command-line program is provided, called **copyBinaryArt**. This program will copy all of the 8 source files associated with a particular character (letter or number) or groups of letters and numbers, but during the copying will replace the first 3 letters of the source filenames with an incrementing 3-digit sequence number. This sequence number will preserve the order of the filenames when they are processed and displayed in the terminal window by your script.

`copyBinaryArt` can be run via the following (which will show how it is used):

```
$ /units/kit213/assignment/copyBinaryArt
```

```
Usage: copyBinaryArt art-source art-destination starting-sequence-number string
   art-source: the source directory containing the original decimal-named files
   art-destination: the destination directory to copy the decimal-named files to
   starting-sequence-number: starting number used for first file created, incremented for each file
   string: the string to be converted to 8-bit decimal-named files
```

For example, to copy the source art files to represent the text "Hi", the command would be: (assuming you have created a subdirectory in your current directory called "**`myDir`**")

```
$ /units/kit213/assignment/copyBinaryArt /units/kit213/assignment myDir 1 "HI"
    Source directory /units/kit213/assignment verified to exist!
    Destination directory myDir verified to exist!
    The starting sequence number is 1
    The string to convert is:  HI
    Copying files for the character 'H':
    Copying files for the character 'I':
```

After the copy has completed, the contents of **`MyDir`** would be:

001-0.grf 002-198.grf 003-198.grf 004-198.grf 005-254.grf 006-198.grf 007-198.grf 008-198.grf	*Files that represent the letter "H"*

`009-0.grf`
`010-252.grf`
`011-48.grf`
`012-48.grf`
`013-48.grf`
`014-48.grf`
`015-48.grf`
`016-252.grf`
	Files that represent the letter "I"

The ultimate task for your script to complete then is to process all of the files that are contained in your local directory, converting the decimal part of each filename to binary, replacing zeros in the resulting binary values with spaces, and replacing ones with some prominent character (like an X) before displaying the result on the terminal screen. Bear in mind when your script is assessed, the marker will use completely different source files to test your script.

> ➢ You are free to create your own additional artwork (using 8 filenames to represent an individual 8 by 8 grid), but your script (see below) must be able to process any validly named file that includes the naming convention described above i.e. 3-digit sequence number, dash, decimal value (0 up to 255), then a file extension (e.g. `.grf`).

5.4 Source filename validity

Because of poor quality control from the art department, occasionally some files may be present in the local source directory that is used for processing (and thus testing by the marker) that do not correctly conform to the file-naming scheme. Your script will have to correctly process these files and correctly categorise (and move) them to another directory. If you have copied some valid source files to your local directory, then you will also need to manually create some invalidly named files to test your script correctly.

5.4.1 Valid filenames

Filename format: *`XYX-decimal.ext`*

- *`XYX`* : a 3-digit sequence code
- `-` : the dash character
- *`decimal`* : a numeric value between 0 and 255 inclusive
- *`.ext`* :a file extension such as .grf or .doc, .xls etc

Example: any of the following would be considered potentially valid filenames in your local source directory (the validity will also depend on which particular filename extension has been specified)

```
001-65.grf 369-0.txt 501-127.doc 999-255.xls
```

5.4.2 Invalid filenames

Anything not fitting the patterns described above is **invalid**. Your script will need to categorise each invalid source filename as detailed below (in order of priority), and then **copy the invalid source file** to a specific subdirectory of your destination directory (see Task Requirements later).

- **INVALID1** - one or more non-numeric characters where the decimal value should be.
- **INVALID2** - numeric-only decimal values, but the decimal value is more than 255.
- **INVALID3** - one or more non-numeric characters where the sequence code should be.
- **INVALID4** - numeric-only sequence codes, but there are more than three digits.
- **INVALID5** - numeric-only sequence codes, but there are less than three digits.
- **INVALID6** - no sequence code or no decimal part.
- **INVALID7** - no file extension, or the extension is different to the argument to the script.

Examples of filenames for each category:

Category	**Example invalid filenames (*assume extension specified is .grf*)**		
INVALID1	001-ba0.grf	001-bad.grf	123-12*.grf
INVALID2	123-256.grf	999-999.grf	111-1234.grf
INVALID3	a-000.grf	01x-111.grf	1q7-127.grf
INVALID4	1222-122.grf	11999-100.grf	99999-254.grf
INVALID5	12-122.grf	1-100.grf	99-254.grf
INVALID6	wrong.grf	-123.grf	123-.grf
INVALID7	badfile	123-100.txt[1]	911-000.doc[1]

6. Task Requirements (this is what your script must do)

1. Your script for this task **must** be named `displayBanner.sh`
   If your script has a different name, it will not be assessed.

2. Make sure your script is not unnecessarily complex - your script should use **consistent indentation** and include **whitespace/blank lines** (where relevant) to make the script more logical and easier for a person to read. You must also include **general inline comments** in the script code to indicate the purpose of more complex command combinations etc.

3. Your script must start with the following first line:

   ```
   #!/bin/sh
   ```

   (*this specifies the shell to be used to interpret the rest of the commands in the script*)

4. Your script **must include comments** at the beginning (near the top of the script) to specify:
   a. the script's purpose,
   b. the author (including student number),
   c. and the date (when last modified).

[1] *File extension doesn't match the one specified as a command-line argument!*

5. Your script **must accept 3 command-line arguments** provided when the script is run:
    - Argument 1 – the local source directory containing the files to be processed
    - Argument 2 – the local destination directory that will be used to copy invalid filenames to
    - Argument 3 – the file extension (without a leading dot) to be used for valid source files (eg `grf`)

    The directory names and the file extension that are provided by arguments 1,2 and 3 must not be 'hardcoded' (specified literally) in your script, i.e., their value must come from the command line arguments. If any of the 3 arguments are missing, the script should provide a usage warning message and then exit – **the user must not be prompted to enter missing values when the script is running**. See ***Example Output***.

6. Near the beginning of your script, **you need to check** for each of the directories associated with the arguments 1 and 2 (the source directory and the destination directory):

    a. Exists. If the directory does not exist, your script must display an error message and then it should exit. The directory name provided must use a relative path – see ***Example Output*** for examples.

    b. Is readable, executable (for both directories) and writeable (for the destination).
        i. If the directory is not readable, your script must display an error message and then it should exit.
        ii. If the directory is not writeable, your script must display an error message and then it should exit.
        iii. If the directory is not executable, your script must display an error message and then it should exit.

7. For **every file** in the specified source directory, the script must:
    a. For a **validly named** file:
        i. Extract the **decimal value** from the filename, **convert it to 8-bit binary**, and then output to the terminal window the binary value but with every 0 replaced with a space character (" ") and every 1 replaced with an "X" character. *The following is an example of the decimal value 59 (00111011_2) displayed with the space character shown below as a gray block ░ for clarity purposes only):*

```
░░XXX░XX
```

    b. For an **invalidly named file** that does not match the validity test:
        i. display the following output to the terminal screen **after all valid files have been processed** (i.e defer the output to the end of processing):
            - *X* refers to the specific invalid category (see section 5.4.2)

- *filename* refers to the original source filename
- *invalid-description* refers to the text defined at the end of this step)

```
INVALIDX - filename (invalid-description)
```

Category	*invalid-description*
INVALID1	`non-numeric decimal part`
INVALID2	`decimal part too large`
INVALID3	`invalid sequence code - non-numeric characters`
INVALID4	`invalid sequence code - too many digits`
INVALID5	`invalid sequence code - too few digits`
INVALID6	`no sequence code or decimal part`
INVALID7	`filename wrong or missing/wrong file extension`

ii. Move the invalid file to a subdirectory of the local destination directory that was specified as the second argument to the script, and call the subdirectory **INVALIDX** (the script should only create the subdirectory if it does not already exist and also only when a file in this category is found).

8. If the source directory specified as the first argument to the script contains no **files**:
   a. display the following output to the terminal screen:
      - *sourcedir* refers to the directory name specified in argument 1

```
The directory sourcedir contained no files…
```

   b. exit the script

9. If the source directory specified as the first argument to the script contained **files**:
   a. display the following output to the terminal screen as the very last line of output:
      - *filecount* refers to the number of files processed by the script

```
filecount files were processed
```

   b. exit the script

> ➢ You must also provide (submit) some specific output from testing your script on a specific sequence of test files – more details are given in the next section.

7. Test output for submission

Once your script is ready for submission, you must also include some specific test output. The easiest way to do this is to make a *typescript* (log) of the terminal window output. When ready, create a subdirectory for some specific art files – eg **`submitSource`**

7.1 Create a new typescript:

```
$ script
Script started, file is typescript
```

7.2 Copy artwork files

Copy the art files for your student number and first and last names to the submitSource directory, for example, if your student number is 123456 and your name is John Smith, you would use:

```
$ /units/kit213/assignment/copyBinaryArt /units/kit213/assignment submitSource 100
"123456 John Smith"

Source directory /units/kit213/assignment verified to exist!
Destination directory submitSource verified to exist!
The starting sequence number is 100
The string to convert is 123456 John Smith
Copying files for the character '1':
Copying files for the character '2':
etc…
```

7.3 Run test output

```
$ ./displayBanner.sh submitSource submitDest grf
```

(output shown in 3 columns for brevity)

```
  xx
xxxx
  xx
  xx
  xx
  xx
xxxxxx

 xxxxx
xx   xx
xx  xxx
  xxxx
 xxxx
xxx
xxxxxxx

xxxxxxx
    xx
   xx
  xxxx
     xx
xx   xx
 xxxxx

   xxx
  xxxx
 xx xx
xx  xx
xxxxxxx
    xx
    xx

xxxxxx
xx
xxxxxx
     xx
     xx
xx  xxx
 xxxxx

  xxxx
 xx
xx
xxxxxx
xx   xx
xx   xx
 xxxxx
```

```
   xx
   xx
   xx
   xx
   xx
xx xx
 xxx

 xxxxx
xx   xx
xx   xx
xx   xx
xx   xx
xx   xx
 xxxxx

xx   xx
xx   xx
xx   xx
xxxxxxx
xx   xx
xx   xx
xx   xx

xx   xx
xxx  xx
xxxx xx
xx xxxx
xx  xxx
xx   xx
xx   xx
```

```
 xxxxx
xx   xx
xx
 xxxxx
     xx
xx   xx
 xxxxx

xx   xx
xxx xxx
xxxxxxx
xx x xx
xx x xx
xx   xx
xx   xx

xxxxxx
  xx
  xx
  xx
  xx
  xx
xxxxxx

xxxxxxx
  xxx
  xxx
  xxx
  xxx
  xxx
  xxx

xx   xx
xx   xx
xx   xx
xxxxxxx
xx   xx
xx   xx
xx   xx

136 files were processed
```

7.4 End typescript

End the typescript by pressing control-D on the keyboard:

```
Script done, file is typescript
```

You will then have a file called "typescript" containing the output captured – submit this file.

8. Algorithm Hints

The suggested approach is you take the decimal part from the filename and then iteratively divide the value by decreasing powers of 2 (starting with 128), working out the remainder, and repeating by dividing the next (lower) power of 2 into the remainder (this was shown in a lecture and some practical exercises called *comparison with descending powers of two approach*).

Two expression operators to convert decimal to binary that may be useful:

- expr x / y (*divide x by y*)
- expr x % y (*get the remainder from x divided by y, % is the modulus operator*)

Example – to convert the decimal value 62 to 8-bit binary:

- 1st binary digit = 62 / 128 = **0**; remainder is 62 % 128 = 62
- 2nd binary digit = remainder (62) / 64 = **0**, remainder is 62 % 64 = 62
- 3rd binary digit = remainder (62) / 32 = **1**, remainder is 62 % 32 = 30
- 4th binary digit = remainder (30) / 16 = **1**, remainder is 30 % 16 = 14
- 5th binary digit = remainder (14) / 8 = **1**, remainder is 14 % 8 = 6
- 6th binary digit = remainder (6) / 4 = **1**, remainder is 6 % 4 = 2
- 7th binary digit = remainder (2) / 2 = **1**, remainder is 2 % 2 = 0
- 8th binary digit = remainder (0) / 1 = **0**, remainder = 0 % 1 = 0

The result is 00111110. Your algorithm should implement this but by using iteration (looping).

Another tricky component you may find slightly more challenging is determining which character(s) in a valid filename are part of the decimal part. This is because the decimal part may be 1, 2 or 3 characters long. One suggested approach includes using the **`basename`** command (introduced here).

`basename` can strip the extension (called a suffix) from a filename.

You can determine the filename without the file extension/suffix part via:

```
file=some method of getting the current whole filename to be processed…
extention=the extension provided as the 3rd argument to the script
filebase=`basename "$file" .$extension`
```

e.g. if **`file`** is **`123-254.grf`**, **`extension`** is **`grf`**, then **`filebase`** would be **`123-254`**

It is then easier to determine the decimal part by using the filename with the extension removed, and other commands (such as **cut**)

9. Example output

9.1 Example files

You of course are free to create any directory you like in your **kit213script** directory and populate it with files for testing purposes (this is highly recommended – you need to test your script works correctly!). The marking process will **not** use the filenames or directory name that are listed below, marking will use a different directory and different testing files.

In this example, assume **kit213script** is the current working directory, and subdirectories called **exampleSource** and **exampleDest** have been created inside **kit213script**. The contents of **exampleSource** are the following (invalid) files:

Filename
000-256.grf
001-ba0.grf
001-bad.grf
0-128.grf
02-128.grf

Filename
9999-128.grf
a-000.grf
badfile
000-255.doc
wrong.grf

The source files for the art for the letter "A" have also been copied to **exampleSource**:

```
$ /units/kit213/assignment/copyBinaryArt /units/kit213/assignment exampleSource 100 "A"
Source directory /units/kit213/assignment verified to exist!
Destination directory exampleSource verified to exist!
The starting sequence number is 100
The string to convert is:  A
Copying files for the character 'A':
```

The following is a sample output of the script you must develop. The text up to and including the **$** is the shell prompt:

9.2 Example of successful run

```
$ ./displayBanner.sh exampleSource exampleDest grf

  XXX
 XX XX
XX   XX
XX   XX
XXXXXXX
XX   XX
XX   XX

INVALID7 - 000-255.doc filename wrong or missing/wrong file extension
INVALID2 - 000-256.grf decimal part too large
INVALID1 - 001-ba0.grf non-numeric decimal part
INVALID1 - 001-bad.grf non-numeric decimal part
INVALID5 - 0-128.grf invalid sequence code - too few digits
INVALID5 - 02-128.grf invalid sequence code - too few digits
```

```
INVALID4 - 9999-128.grf invalid sequence code - too many digits
INVALID3 - a-000.grf invalid sequence code - non-numeric characters
INVALID7 - badfile filename wrong or missing/wrong file extension
INVALID6 - wrong.grf no sequence code or decimal part
18 files were processed
```

The contents of **exampleDest** after the script has run are:

```
exampleDest /:               exampleDest /INVALID2:    exampleDest /INVALID5:
INVALID1                     000-256.grf               0-128.grf
INVALID2                                               02-128.grf
INVALID3                     exampleDest /INVALID3:
INVALID4                     a-000.grf                 exampleDest /INVALID6:
INVALID5                                               wrong.grf
INVALID6                     exampleDest /INVALID4:
INVALID7                     9999-128.grf              exampleDest /INVALID7:
                                                       000-255.doc
exampleDest /INVALID1:                                 badfile
001-ba0.grf
001-bad.grf
```

9.3 Examples showing error messages

Missing one or more command-line arguments

```
$ ./displayBanner.sh
Usage: ./displayBanner.sh source-directory destination-directory file-extension
```

Source directory does not exist

```
$ ./displayBanner.sh badSource exampleDest grf
Source directory badSource not found
```

Destination directory does not exist

```
$ ./displayBanner.sh badSource exampleDest grf
Destination directory exampleDest not found
```

Filename extension does not match the majority of files

```
$ ./displayBanner.sh exampleSource exampleDest bad

INVALID7 - 100-0.grf filename wrong or missing/wrong file extension
INVALID7 - 101-48.grf filename wrong or missing/wrong file extension
INVALID7 - 102-240.grf filename wrong or missing/wrong file extension
...
```

[truncated – all files with an extension that does not match will be category INVALID7]

Source directory not readable or executable

```
./displayBanner.sh exampleSource exampleDestination grf
Source directory exampleSource is either not readable or executable
```

Destination directory not readable or executable or writeable

```
$ ./displayBanner.sh exampleSource exampleDestination grf
Destination directory exampleDestination is either not readable, executable or
writable
```

10 Submitting Your Assignment

- The instructions that follow assume you have created a directory called **`kit213script`** in your **`ictteach`** home directory, your script file is called **`displayBanner.sh`** and it is located inside the **`kit213script`** directory, and if you successfully completed all requirements, you will also have a file called typescript that contains test output.
- These instructions also assume you are either using a lab macOS-based computer (such as those in the networks labs), or you are using your personal computer. If you are off-campus, you must be running GlobalProtect VPN to be able to connect to ictteach via SSH and SCP.

All assignment submissions are through MyLO. Submitting your assignment to MyLO requires you to first make a compressed copy of your script file on **`ictteach`**, copy that compressed file from **`ictteach`** to your local computer, and then upload the compressed file from your local computer to the MyLO submission area. Please read and follow the instructions below carefully – ask your tutor if you have any difficulty, and we suggest you try the instructions well-ahead of the due date so you do not encounter any last minute problems.

10.1 Start on ictteach

You must first create a compressed version (a copy) of your assignment script on **`ictteach`**, and then copy this version to your local computer:

On **`ictteach`**, run the following commands (**$** is the prompt):

```
$ cd ~/kit213script
$ tar -cvf kit213assignment.tar displayBanner.sh typescript[2]
$ gzip -c kit213assignment.tar > kit213assignment.tar.gz
```

You then need to copy the **`kit213assignment.tar.gz`** file to your local computer – instructions on how to do this differ depending on what local operating system your local computer is using – see below. The instructions for each operating system use a command called **`scp`** (secure copy), which uses the SSH protocol behind the scenes to copy files.

10.2 Apple macOS (using terminal)

If you are using an Apple macOS-based computer to submit, copy your compressed script file to the local computer via:

a) From a new terminal window (i.e. not a terminal session already connected to **`ictteach`**), type the following (**%** is the prompt, do not type that)

```
% scp  username@ictteach.its.utas.edu.au:kit213script/kit213assignment.tar.gz  ~/Desktop
```

(*replace* ***username*** *with your own UTAS username (this is NOT your email address!)*)

[2] If you did not create the typescript file, simply leave the "typescript" filename out

b) You should then be prompted for your **ictteach** password:

```
username@ictteach.its.utas.edu.au's password: 
```

c) After typing your password (it will not be shown onscreen), if you have correctly authenticated, you should get a notification line like the following to indicate the file has been downloaded (to your Desktop in this instance – the file size and data transfer rate shown will differ to your file size and data transfer rate)

```
kit213assignment.tar.gz          100% 3727     2.6MB/s   00:00
```

10.3 Microsoft Windows (using putty)

If you are using your own Microsoft Windows-based computer to submit, we assume you have previously installed the **putty** program to access **ictteach**. Putty also includes some other programs when you install it – **pscp.exe** is one of them and it is the program needed. If you are not using putty but have been using some other access method, you will need to investigate how to use the **scp** command for that method.

Copy your compressed script file to the local computer via:

a) Start a new *command prompt* window (select the Windows start menu icon and then type **cmd.exe**).

b) In the command prompt window, use the following command to navigate to the putty local installation directory (in the example, putty is installed in a putty subdirectory of the standard location, **c:\Program Files** – your local putty directory may differ). (**C:\Users\username>** is the prompt, do not type that):

```
Microsoft Windows [Version 10.0.19042.1110]
(c) Microsoft Corporation. All rights reserved.

C:\Users\username> cd c:\program files\putty
```

c) If successful, the current directory should now be the putty install directory. **pscp.exe** is putty's version of the scp command, and it should be installed here. Run the following command (**C:\Program files\PuTTY>** is the prompt, do not type that):

```
C:\Program files\PuTTY> .\pscp.exe
username@ictteach.its.utas.edu.au:kit213script/kit213assignment.tar.gz
"%userprofile%"
```

(*Replace* **username** *with your UTAS username (this is NOT your email address)*)

➢ Note that there are only two spaces in the above command – between **pscp.exe** and your ***username***, and between **.gz** and "**%userprofile%**". All parts are on the same line, it just appears word-wrapped over lines in this document.

d) You will be prompted for your **`ictteach`** password (the password will not appear on screen):

```
username@ictteach.its.utas.edu.au's password:
```

If you have typed the commands correctly, your compressed script file, **`kit213assignment.tar.gz`** should now be in your Windows profile home location (usually c:\users\`username`) but your path may differ.

10.4 MyLO submission (all students):

If you have successfully copied your compressed **`kit213assignment.tar.gz`** script file from **`ictteach`** to your local computer, it should now be on your local system. You can now submit this file to MyLO:

a) Navigate through the KIT213 MyLO site's top menu to the submission link:
Assessments → Assignments → UNIX Shell Scripting
b) Scroll down and select the **Add a File** button.
c) Chose the **My Computer** option, and then choose the **Upload** button. Navigate to your local location and select the **`kit213assignment.tar.gz`** file that you copied earlier and finally choose **Open**.
d) Back in the **Add a File** dialog window, choose the **Add** button at the bottom.
e) Then, if everything appears to be ok (verify you can see your file listed) choose the **Submit** button.

10.4.1 Revised submission?

If it is before the due date and time, follow all the submission steps again to make a compressed copy of your script, download the copy to your computer and then upload your copy to MyLO. We will only assess the latest submission.

10.4.2 Late submission?

If your assignment is late then you should submit your compressed script file to MyLO as above – the University's new assessment policy means you will lose 5% of your score for every day the submission is late – assignments will not be accepted after 10 days past the scheduled submission date.

10.4.3 Need Help?

You are encouraged to seek assistance from your lecturer or tutors in practicals or during consultation **after you have seriously thought about the assignment**. Please note that we can provide general advice – but you are expected to write and debug (correct) your own code.

When writing your script, think about what steps you need to do, don't try to write the code all at once. Implement a small part first, test that it is working correctly, then implement a bit more, repeating the process of thinking, writing (coding), testing and refining/fixing. Do not leave the development of your script too late – you will run out of time!

10.4.4 Extra Hints:

When testing your script it's recommended you "clean up" any temporary files or directories already created so you can verify your script correctly creates directories when required. You might like to write another simple setup script that you can keep quickly reuse to recreate your testing log directory and populate it with test files prior to each time you try to run your assignment script. A reminder – the **`touch`** command can be used to quickly create an empty file if it doesn't already exist e.g. **`touch 123-111.grf`**

11 Plagiarism

Plagiarism is a very serious matter, and ignorance of this is not an excuse. If you submit work claiming it to be your own, and it is not original work of your own, this is cheating. This means for example that you cannot submit sections of code that have been written by other students or sourced from the Internet and claim you wrote the code yourself. Plagiarism can result in academic penalties both in relation to your assignment, and also on your permanent university record. *As an example, you cannot take code written by someone else and change variable names, comments and spacing to make it appear you wrote the code – this would still be considered plagiarism.*

KIT213 Scripting Assignment Marking Rubric

Criteria	HD	DN	CR	PP	NN
1 Script Execution (5%)	The script runs without any syntax errors or unusual output	The script runs without any syntax errors, but some minor unusual output occurs	The script runs without any syntax errors, but some significant unusual output occurs	The script runs with very minor syntax errors that do not prevent it from continuing	The script does not run, or it has significant syntactic errors, or the script contents are trivial
2 File source and destination directories (5%)	The source and destination directories are provided as command line arguments, they are verified to exist, and all the directory permissions are correctly verified	The source and destination directories are provided as command line arguments, they are verified to exist, and some of the directory permissions are correctly verified	The source and destination directories are provided as command line arguments, they are verified to exist, but none of the directory permissions are correctly verified	The source and destination directories are provided as command line arguments, but no verification is performed for existence or permissions	The source and/or destination directories are not provided as command line arguments (and/or they are hardcoded in the code)
3 File source directory contents (5%)	Script correctly outputs when the source directory is empty, and it correctly counts the number of files processed. Only syntax and commands discussed in classes have been used			Script correctly outputs when the source directory is empty. Only syntax and commands discussed in classes have been used	Script does not correctly output when a directory contains no files
4 Decimal to binary conversion (25%)	The script correctly finds and converts the decimal part of all **valid** filenames to their binary equivalent. Only syntax and commands discussed in classes have been used		The script correctly finds and converts the decimal part of some **valid** filenames to their binary equivalent. Only syntax and commands discussed in classes have been used	The script correctly finds and converts the decimal part of all filenames (whether valid or invalid) to their binary equivalent	No correct decimal to binary conversion is evident
5 Binary output (20%)	The script correctly converts the 8 binary values associated with an 8x8 character representation to the output representation and displays the result in the output. Only syntax and commands discussed in classes have been used			The script displays the 8 binary values associated with an 8x8 character representation to the output in the correct order. Only syntax and commands discussed in classes have been used	The script does not display any binary representation to the output
6 Invalid files categorisation (15%)	All Invalid files are correctly identified in the output, and they are moved to the corresponding category subdirectory. If not initially present, the corresponding category subdirectory is only created when and if files in this category are discovered. Only syntax and commands discussed in classes have been used	All Invalid files are correctly identified in the output, and they are moved to the corresponding category subdirectory. The corresponding category subdirectory is created even if no files in the category are found. Only syntax and commands discussed in classes have been used	Some Invalid files are correctly identified in the output, and they are moved to the corresponding category subdirectory, but some files are misclassified	Some Invalid files are correctly identified in the output, but they are not moved to the corresponding category subdirectory, and some files are misclassified	No invalid files in this category are correctly identified, nor are they moved
7 Script Comments (5%)	Informative script comments are included at the start, as well as throughout the script (where appropriate)	Informative script comments are included at the start, and many other comments are included, but some comments are obviously missing	Informative script comments are included at the start, but there are few other comments - there is an obvious need for other comments	Informative script comments are included at the start, but very limited comments elsewhere	No comments or only extremely sparse or inappropriate comments are included
8 Script Layout (10%)	Excellent, consistent use of indentation and whitespace throughout	Excellent use of indentation and whitespace however there are some minor examples of inconsistency	Good use of indentation and whitespace but a few sections are inconsistent	Occasional, correct use of indentation and whitespace but many sections are inconsistent	No attention provided to layout, major lack of consistent use of indentation and whitespace
9 Typescript output (10%)	The typescript output correctly displays the student's student number and first and last name in the correct representational format			The typescript contains some correct information	Typescript is not included, or it contains incorrect information

Note for criteria grades above NN are only applicable if the script contents are non-trivial