The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 214– Programming Principles and Practice

Assignment 7

Date Due: December 4, 2020

Total Marks: 20

1 Submission Instructions

- Assignments must be submitted using Canvas.
- Create this bash script and run/test it on tuxworld.
- **Do not** be fancy. Keep it simple, and ***only*** use what you were taught in this course.
- Always include the following identification in your solutions: your name, NSID, student ID, instructor's name, course name, and course section number.
- No late assignments will be accepted. See the course syllabus for the full late assignment policy for this class.
- **VERY IMPORTANT**: Canvas is very fragile when it comes to submitting multiple files. We insist that you package all of the files for all questions for the entire assignment into a **ZIP** archive file. This can be done with a feature built into the Windows explorer (Windows), or with the **zip terminal command** (LINUX and Mac). We **cannot accept** any other archive formats. This means no tar, no gzip, no 7zip. Non-zip archives will not be graded. We will not grade assignments if these submission instructions are not followed.
- Instructions on "how to create zip archives" can be found here `https://canvas.usask.ca/courses/9771/pages/how-to-zip-slash-compress-your-files`

2 Background

This background section serves as a "translation" for man pages of commands that can be found in tux-world that weren't explained in the previous assignment(s). Even though this is provided to you it is still recommended that you spend some time looking at the man pages to see the translations for yourself. Additionally, this background section will provide details of any intricacies of `bash` that you may find useful for `bash` scripting.

2.1 Easy to read `man` pages (or at least easier than the ones on tuxworld)

2.1.1 `mkdir(1)`

Short for "make directory". As you should know from the textbook, the `mkdir` command will create a sub-directory. However, this command can take in command options!

`mkdir -p <PATH>` If the directory you are trying to create already exists, do not output an error.

`mkdir -v <PATH>` When a directory is successfully created, print out a message.

2.1.2 `grep(1)`

Short for "global regular expression print". As you should know from the textbook, the `grep` command will print lines when it finds a match to the `pattern` you provided it. In the textbook, the usage example provided was printing out the lines where the `pattern` provided to `grep` was found. And surprise, surprised - this command can also take in command options!

`grep -s <PATTERN> <FILE>` If the file provided does not exist or is not readable, do not show any error messages to `stdout`.

`grep -q <PATTERN> <FILE>` `q` stands for quiet and says to not write anything to `stdout`

`grep -F <PATTERN> <FILE>` Interpret the provided `pattern(s)` as a fixed string, not as regular expressions. Note that this is an upper-case 'F'!

2.1.3 `zip`

If you were to guess the purpose of this command, would you guess that it would compress (create a `.zip`) a provided file? If you did guess that, you're right!

`zip -r <OUT_NAME>.zip <DIRECTORY>` `r` stands for recursive. `zip` will recursively `zip` a provided directory. Which means all of the contents in the provided directory will also be archived as a `.zip` file.

2.2 Miscellaneous `bash` stuff

2.2.1 `IFS`

`IFS` is a special `bash` environment variable that stands for Internal Field Separator. This command is commonly used by the `read` command, parameter expansion, or command substitution. By default `IFS` is set to be `<space><tab><newline>`, which means that if something is being read, any occurrence of a space, tab, or newline will be interpreted as the end of a word. Example, if you have a text file that contains the following:

```
I am
a  word
I
am not a
person
```

And you were to run a script that was:

```
#!/bin/bash

for i in $(cat file.txt)
do
    echo $i
done
```

the `echo` command would output:

```
I
am
a
word
I
am
not
a
person
```

Instead, if you wanted to output on `stdout` the exact contents of the file, line by line, the value of `IFS` would have to be set (before the `for` loop) to equal the newline character, and the newline character alone.

2.2.2 Converting a string to all lower-case or upper-case characters

Previous to `bash` version 4, the way to convert an entire string to all upper- or lower-case characters was with `tr`. In `bash` version 4 (and 5) you can still do this, however new syntax was introduced to more easily do this and is demonstrated below.

```
$ {<VARIABLE>,,}
```

The above specifies that if the provided variable contents is a string, then the entire string should be converted to all lower-case characters.

```
$ {<VARIABLE>^^}
```

The above specifies that if the provided variable contents is a string, then the entire string should be converted to all upper-case characters.

2.2.3 `break`ing out of or `continue`ing iterations of a loop

While performing iterations in a `while` or `for` loop, the commands `break` and `continue` are used to alter the command flow.

The `break` command will end the loop, in other words, you could be iterating from 0 to 6, it doesn't matter if you were on the first iteration or the fifth. If at any point the `break` command is executed whatever remains in the loop will be skipped, the loop will be terminated, and execution will resume after the loop.

The `continue` command will move on to the next iteration of the loop. Similar to the `break` command, if at any point in the loop a `continue` command is executed the remainder of the code in the loop will be skipped, however the loop is not terminated but it will restart at the next iteration.

2.2.4 The double pipe (||)

In bash, the double pipe means OR. In the following example, if <COMMAND1> fails/errors (the exit status is not 0) then perform <COMMAND2>. Otherwise, if the exit status of <COMMAND1> is 0 (the command succeeded) then do not perform <COMMAND2>.

```
$ <COMMAND1> || <COMMAND2>
```

3 Your Task

Question 1 (20 points):

Have you ever had someone ask you for music recommendations based on genre? If so, have you ever gone to your “Liked Songs" in say, Spotify, and tried to filter your songs based on genre but couldn’t? I have. The only solution I’ve been able to think of has been to store song data in a pretty convoluted and kinda wacky way... So here we are.

The Problem

Updated text is in red.

You are provided with one incomplete .sh file, and one .txt file.

asn7q1.sh: This file contains an implemented function to create a directory structure based on music genres, and some miscellaneous variables. The variables declared include variables to store column numbers for `music.txt`, as well as the main genres that are found in `main.txt`. The main genres are indie, pop, punk, and rock, but there can contain more that do not fall under those categories. **Do not edit any pre-existing code in this file, if you do you are guaranteed to lose marks.**

music.txt: This file contains parsed information about selected songs from a Spotify query. Each individual line in this file contains data about an individual song, with the exception of the first line since it contains headers for the data. No song appears twice. **Do not edit this file at all, if you do you are guaranteed to lose marks.** The columns in this file appear as followed:

Entry Number The entry number of the song, this starts at 0 and goes to 513.

Album Type Describes whether the song was released as an: album, single, or compilation.

Album Name Name of the album that the song appears on.

Artist Name of the artist that released the song.

Featured Artists Any additional artists that the main artist collaborated with. Or in the case of remixes, the artist that remixed the song.

Song Name The name of the song.

Genre The genre the artist of the song is classified under. There is only one genre per song, and the genres themselves are not limited to being indie, pop, punk, or rock.

Release Year The year in which this version of the song was released.

Preview URL A link to listen to a preview of the song. (This is a real link and will provide you a 30 second clip of the song.)

You are going to edit the `asn7q1.sh` script to parse through the lines in `music.txt` to create and organize a directory hierarchy. You will also have to populate the appropriate directories with text files.

The Directory Structure

The root directory of this hierarchy should contain a folder for each genre that appears in `music.txt`. The existing function in the given shell script will create all the genre and sub-genre directories needed. You should not have to create any more. The only directories you need to create are for artists and the uncategorized directory.

A sub-folder of `music/` will be created for every genre that appears in the `music.txt` file in the "Genre" column. However, there are four popular genres that have a lot of songs in them, so they are broken down by sub-genre. These genres are **indie, pop, punk, and rock.** Sub-genres are created for any song whose full genre name contains one or more of the substrings `"indie"`, `"pop"`, `"punk"`, or `"rock"`.

Thus, if the genre `"pop rock"` appears in the database, then that song will cause the creation of the sub-genres `"music/pop/pop-rock"` and `"music/rock/pop-rock"` because the genre name contains both `"pop"` and `"rock"`. Or if the genre is `"punk grunge"` then this the creation of only one sub-genre, `music/punk/punk-grunge` because it only contains the substring `"punk"` and none of `"indie"`, `"pop"`, or `"rock"`. If the genre contains none of those substrings, then no sub-genre is created. For example, the all songs in the genre `"blues jazz"` would reside in the main genre directory `music/blues-jazz`.

What is not created by the given function are the artist subdirectories and the song files. Your task will be to go through `music.txt` and create the artist folders and song files for each song. Each genre (and sub-genre) directory will contain more directories named after any music artists that are classified to fall in that genre (which you will create). For example, the first song entry of `music.txt` (line 2) specifies that Weezer falls under the "pop rock" genre. *In this case, a directory for Weezer should be created in both pop/pop-rock, and rock/pop-rock.*. If a song does not have a genre associated with it (it is set to "None") then the artist of that song will be stored under the root directory in a directory called `music/uncategorized`.

The artist directories will contain text files. These text files represent albums and will be named as: the year that the album in which the song appears was released, the album name in which a song appears in, and the album type (i.e. whether it is an album, single, or compilation.) Each line in the album file represents a song that appears in the album and must contain the song name and the link to the preview of the song. Details on *how* to do this follows after a visual example of the above descriptions.

Thus, if a song's genre is `"indie pop"`, and its artist is `<artist>` then you'll need to creqte folders `music/indie/indie-pop/<artist>` and `music/pop/indie-pop/<artist>` (if they don't already exist), and then create a song file in both of those artist directories. If the genre doesn't contain one of the four sub-genred genres, just create the artist folder in the already-created genre folder, for example, if the genere is `"blues jazz"` you'd create `music/blues-jazz/<artist>` and put it's song file in that folder. If the genre is not given, you'd create `music/uncategorized/<artist>` and put the song's file in that directory.

If a song matches one of the sub-genred genres **exactly** do not put it in a sub-genre folder, put it in a main folder. Such as if the genre is "pop", you should put the song file in `music/pop/<artist>`.

A small example of the directory structure:

```
music
├── pop
│   ├── pop-punk
│   │   └── <ARTIST-NAME of song that is classified in the "Pop Punk" genre>
│   │       └── <RELEASE_YEAR>_-_<ALBUM_NAME>_(<ALBUM_TYPE>).txt
│   ├── electropop
│   │   └── <ARTIST-NAME of song that is classified in the "Electropop" genre>
│   │       └── <RELEASE_YEAR>_-_<ALBUM_NAME>_(<ALBUM_TYPE>).txt
│   └── <ARTIST-NAME of song that is classified in the "Pop" genre>
│       └── <RELEASE_YEAR>_-_<ALBUM_NAME>_(<ALBUM_TYPE>).txt
├── punk
│   ├── pop-punk
│   │   └── <ARTIST-NAME of song that is classified in the "Pop Punk" genre>
│   │       └── <RELEASE_YEAR>_-_<ALBUM_NAME>_(<ALBUM_TYPE>).txt
│   └── <ARTIST-NAME of song that is classified in the "Punk" genre>
│       └── <RELEASE_YEAR>_-_<ALBUM_NAME>_(<ALBUM_TYPE>).txt
├── rock
│   └── <ARTIST-NAME of song that is classified in the "Rock" genre>
│       └── <RELEASE_YEAR>_-_<ALBUM_NAME>_(<ALBUM_TYPE>).txt
└── uncategorized
    └── <ARTIST-NAME of song that doesn't have a genre classification>
        └── <RELEASE_YEAR>_-_<ALBUM_NAME>_(<ALBUM_TYPE>).txt
```

```
│
│
└── etc.
```

Complete the following tasks:

1. It it doesn't already exist, create a new root directory called `music`.
2. Change your current working directory to be the `music` directory.
3. Call the provided function to create and organize the genre and sub-genre directories. Pretend that this function has a big black box around it. Do not edit this function. *If* you do decide to look at it, under no circumstances are you to use anything from it if you were not taught it in this course. If you want to know about it, feel free to ask us, but you still won't be able to use it.
4. Read the file `music.txt` line by line - be sure to skip the first line as this is a header line and does not contain any song information. Each line in `music.txt` corresponds to one song (again, with the exception of the first line). For each song you will perform the following:
   (a) **OPTIONAL**: If you wish, create progress output. Print a message after every 50 lines read that X entries (where X is a multiple of 50) have been processed. However! If you do decide to do this make sure that it doesn't negatively impact the readability of your script (or remove it before submitting). If the readability is negatively impacted and your marker has issues reading it, then you may potentially lose marks.

   (b) Extract and format all data from the line (song data) into variables. *Note the following special cases*:

      i. The artist name must be converted to all lower case characters and any spaces must be replaced by −'s (a single dash).

      ii. The genre must be converted to all lower case characters and any spaces must be replaced by −'s (a single dash).

      iii. The featured artist name(s) must have any spaces that occur be replaced by _'s (a single underscore).

      iv. The album name must have any spaces that occur be replaced by _'s (a single underscore).

      v. If there is a featured artist for the track then the song name must be formatted as: `<SONG_NAME>_(ft._<FEATURED_ARTIST(S)>)`

   (c) If the song doesn't have a genre (i.e. it is "none"), and if the directory doesn't already exist, create a directory named "uncategorized" under the root directory. If the song does have a genre associated to it then find the appropriate genre directory, or sub-genre directories. Remember, if a song is categorized as a sub-genre then you will have to find all occurrences of that sub-genre and create artist directories and song files within all of them. For example, is the genre is listed as "pop rock" then there exists two directories for this genre, `music/pop/pop-rock` genre, and `music/rock/pop-rock`.

   (d) If it doesn't already exist, create the artist directory in the appropriate genre directory (or directories) . Be careful here, as alluded to in the above directory hierarchy example, there are some songs that are classified as a main genre (such as pop) that should **only** be saved in the pop directory and **not** in any of the genre's sub-genre sub-directories.

   (e) In the appropriate artist directory, write a single line of data to a text file. The name of said text file and line you are writing to it should be formatted similarly as it is in the example provided below. The formatting doesn't have to be identical, but the name and contents **must** contain all of the same data as in the example provided. Furthermore, the text file name must not contain any spaces, instead they should contain _ (a single underscore) where a space would be.
5. Once you have created and populated the music directory create a zip file, saving it as `music.zip`. This zip file should contain an archive for the root directory `music`, along with *every sub-directory* all while **maintaining** the directory hierarchy. This means if you are to unzip `music.zip` then it would have the exact same directory hierarchy as it was before it was zipped. Redirect the stdout

of your zip command to a file called zip-output.txt. *It is important to redirect the output to a text file and not just completely quiet it so that you can see whether or not things failed or succeeded.* The output text file will be handed in separate from `music.zip`, so make sure that you don't accidentally zip it.

Sample Output

This script should not output anything, other than an optional progress update. However, a small snippet of what should be contained in your directory:

```
music
├── pop
│   ├── pop-punk
│   │   └── a-day-to-remember
│   │       └── 2019_-_Degenerates_(single).txt
│   └── christina-aguilera
│       ├── 2010_-_Bionic_(album).txt
│       ├── 2010_-_Bionic (Deluxe_Version)_(album).txt
│       └── 2018_-_Fall_In_Line_(single).txt
├── punk
│   └── pop-punk
│       └── a-day-to-remember
│           └── 2019_-_Degenerates_(single).txt
├── progressive-bluegrass
│   └── ...
└── etc.
```

The contents of `2019_-_Degenerates_(single).txt` is:

```
Degenerates: https://p.scdn.co/mp3-preview/6c137edb6ebe31af1fe00d053cc7b4588bc6178a
```

Note that the above file was created twice in the above example as the genre for A Day to Remember is pop punk.

The contents of `2010_-_Bionic_(Deluxe_Version)_(album).txt` is:

```
Bionic: https://p.scdn.co/mp3-preview/dd134c5385dea09758957d8aef0ab072f12a5e92
Glam: https://p.scdn.co/mp3-preview/33a28ebd49fcab7f4087f83344e3ff69cfcf48c1
Prima Donna: https://p.scdn.co/mp3-preview/048bf03e08a378789baac621674d43ffe9bcc2d3
Sex for Breakfast: https://p.scdn.co/mp3-preview/90afdd7eba4168db54529de5d9892043e9ed9965
Vanity: https://p.scdn.co/mp3-preview/a71d5d436afa6b214bf628acd1ba445327238122
Monday Morning: https://p.scdn.co/mp3-preview/a51b283faefdd10a50253a4b33a2d8750a844c8b
Bobblehead: https://p.scdn.co/mp3-preview/116dcf029bb3c963afcf9d394283466f002c03bf
I Am (Stripped): https://p.scdn.co/mp3-preview/aa874a07b4ca8b55ee3a397bc5f5d28f486d16f2
```

Note that the above file was only created once, as the category for Christina Aguilera is pop.

4 Files Provided

asn7q1.sh A bash script you will be adding to.

music.txt A text file that is tab separated and contains various information about music from Spotify. A single song's data is stored on each line of the file, with the exception of the first line since it is a header line.

5 What to Hand In

Hand in a `.zip` file archive which contains the following files:

asn7q1.sh A bash script that you were instructed to complete without editing any of the pre-existing variables or function/code.

music.zip The zipped contents of the populated music directory that you will have made with your script. Yes, a zip within a zip.

zip-output.txt The output of the command you used to zip your created, organized, and populated music directory.

music.txt The provided text file - unaltered.

6 Grading Rubric

The grading rubric can be found on Canvas.