Project #04: Divvy Ride Analysis

Complete By: Thursday October 24th @ noon
Assignment: F# program to perform Divvy ride analysis
Policy: Individual work only, late work *is* accepted (up to 24 hours late for 10% penalty)
Submission: via GradeScope

Divvy Bike Sharing

You've probably seen the blue bikes that are part of Divvy bike sharing system here in Chicago. The Divvy system is a commuting service where users pay a yearly fee ($75?) to use the bikes for free as long as rides are < 30 minutes in duration. Longer rides incur a surcharge. It's a very popular system, with stations throughout the city and some suburbs.

Interestingly, the Divvy folks also make their ride data available as part of Chicago's open data portal; Divvy's data is available @ https://www.divvybikes.com/data.

The goal in this assignment is to input ride data for a period of time, and perform some analysis of the data: # of bikes, which stations are in use the most often, the average ride duration, etc. Here's a screenshot of the program analyzing the **divvy1.csv** data file:

- 500 rides
- 99 bikes
- Number of rides for an individual bike
- Total time individual bike was checked out
- Average time spent on trips for that bike
- Number of rides to an individual station
- Average time spent on trips leading to that station

```
filename> divvy01.csv

# of rides: 500

# of bikes: 99

BikeID> 102

# of rides for BikeID 102: 6

Total time spent riding BikeID 102: 44 minutes 19 seconds

Average time spent riding BikeID 102: 443.17 seconds

StationID> 20

# of rides to StationID 102: 22

Average time spent on trips leading to StationID 20: 579.6818182 seconds
```

- Histogram of when bikes are borrowed (day of week)
- Output the top 10 most popular station destinations

The program starts by inputting the filename, and then proceeds to input and analyze the
data. A skeleton F# program is provided with the code to open the data file and input the data. Your job is to add the analysis.

When computing the histogram of day trip was started, the day of the week starts at 0 for Sunday and wraps around to 6 for Saturday. The number of stars to output is the number of trips taken /10. The display should include even spacing at the start and a number of stars followed by the actual number at the end.

For example, for the 78 trips started on Sunday in the first data set, the output is:
Sunday: ******* 78

The top 10 stations should print out the station id and number of rides for each of the top 10 most popular stations as destinations by number of rides.

You can solve recursively, or using a higher-order approach (I'd recommend a mix of both). You cannot use loops, no mutable variables, and do not change the data structure.

```
Number of Trips on Sunday: 78
Number of Trips on Monday: 89
Number of Trips on Tuesday: 84
Number of Trips on Wednesday: 93
Number of Trips on Thursday: 74
Number of Trips on Friday: 47
Number of Trips on Saturday: 35

0: ******* 78
1: ******** 89
2: ******** 84
3: ********* 93
4: ******* 74
5: **** 47
6: *** 35
```

```
# of rides to station 1: 32
# of rides to station 0: 18
# of rides to station 6: 14
# of rides to station 3: 13
# of rides to station 5: 13
# of rides to station 2: 12
# of rides to station 14: 12
# of rides to station 15: 11
# of rides to station 7: 10
# of rides to station 8: 10
```

File format

The input file consists of at most 10,000 "rides", representing a snippet of data. The files are not large, which allows the use of recursion without the risk of stack overflow. The file is in CSV format ("comma-separated values"), and each line of the file represents one bike ride. Here's an example:

```
15,22,141,17,5,1124
29,15,181,8,1,1403
21,25,169,12,3,572
...
```

Each line contains exactly 7 values, in this order:

- From station id: integer, id of station where bike was checked out / ride started
- To station id: integer, id of station where bike was returned / ride ended
- Bike id: integer
- Starting hour: the hour, in military time, of when the ride started (e.g. 0 => midnight and 23 => 11pm)
- Starting day: 0 for Sunday, 1 for Monday, 2 for Tuesday, … 5 for Friday, 6 for Saturday
- Trip duration: integer, in seconds

For testing purposes we are providing two input files: "**divvy01.csv**" and "**divvy02.csv**". These files are already installed in the Codio project; if you are working outside of Codio, you can download from the box page under "Projects", "project04-files".

```
filename> divvy02.csv

# of rides: 5114

# of bikes: 1945

BikeID> 2411

# of rides for BikeID 2411: 3

Total time spent riding BikeID 2411: 23 minutes 4 seconds

Average time spent riding BikeID 2411: 461.33 seconds

StationID> 320

# of rides to StationID 2411: 16

Average time spent on trips leading to StationID 320: 595.8125 seconds

Number of Trips on Sunday: 756
Number of Trips on Monday: 693
Number of Trips on Tuesday: 663
Number of Trips on Wednesday: 771
Number of Trips on Thursday: 737
Number of Trips on Friday: 780
Number of Trips on Saturday: 714

0: ********************************************************************************* 756
1: ************************************************************************ 693
2: ******************************************************************* 663
3: ********************************************************************************** 771
4: ************************************************************************* 737
5: ********************************************************************************** 780
6: ******************************************************************** 714

# of rides to station 2: 16
# of rides to station 3: 16
# of rides to station 31: 12
# of rides to station 4: 11
# of rides to station 36: 11
# of rides to station 52: 9
# of rides to station 77: 9
# of rides to station 106: 8
# of rides to station 117: 8
# of rides to station 5: 7
```

Getting Started

To help you get started, we are providing an F# console program that opens the input file, parses the CSV data, and builds a list of lists. This code is provided (along with sample input files) in Codio as well as the course web page ("Projects", "project04-files"). Here's the main program as given:

```
//
// F# program to analyze Divvy daily ride data.
//
// << YOUR NAME HERE >>
// U. of Illinois, Chicago
// CS 341, Fall 2019
// Project #04
//

#light

module project04

//
// ParseLine and ParseInput
//
// Given a sequence of strings representing Divvy data,
// parses the strings and returns a list of lists.  Each
// sub-list denotes one bike ride.  Example:
//
//   [ [15,22,141,17,5,1124]; ... ]
//
// The values are station id (from), station id (to), bike
// id, starting hour (0..23), starting day of week (0 Sunday-6 Saturday)
// and trip duration (secs),
//
let ParseLine (line:string) =
  let tokens = line.Split(',')
  let ints = Array.map System.Int32.Parse tokens
  Array.toList ints

let rec ParseInput lines =
  let rides = Seq.map ParseLine lines
  Seq.toList rides


[<EntryPoint>]
let main argv =
  //
  // input file name, then input divvy ride data and build
  // a list of lists:
  //
  printf "filename> "
  let filename = System.Console.ReadLine()
  let contents = System.IO.File.ReadLines(filename)
```

```
let ridedata = ParseInput contents

//printfn "%A" ridedata

let N = List.length ridedata
printfn ""
printfn "# of riders: %A" N
printfn ""

0
```

Notice that the **main** function inputs the filename, then opens the file and inputs the data. Each line of the file --- one ride --- is turned into a list of 6 items as described earlier. So the resulting data structure is a list of lists. If you uncomment the line denoted above, you can see the format of the input after parsing. Here's what you'll see working with the smaller input file **divvy1.csv** (at least this is the start of the output you'll see):

```
filename> divvy01.csv
[[15; 22; 141; 17; 5; 1124]; [29; 15; 181; 8; 1; 1403];
 [21; 25; 169; 12; 3; 572]; [13; 10; 154; 10; 6; 228]; [6; 14; 156; 8; 0; 680];
 [25; 29; 198; 18; 5; 1234]; [12; 4; 191; 19; 0; 275]; [27; 13; 127; 10; 5; 550];
 [2; 14; 170; 21; 2; 424]; [18; 22; 162; 17; 2; 2655]; [5; 4; 192; 16; 6; 345];
 [4; 10; 185; 17; 3; 1252]; [9; 8; 187; 7; 4; 229]; [8; 28; 133; 17; 2; 1099];
 [0; 0; 191; 17; 2; 976]; [5; 25; 143; 17; 0; 448]; [0; 22; 145; 19; 1; 298];
 [10; 23; 108; 16; 5; 1546]; [11; 1; 121; 11; 4; 144]; [4; 11; 187; 18; 3; 139];
```

You are free to use whatever F# programming environment you prefer; submissions will be collected using Gradescope. If you program using Codio, the environment and console app is already setup for you. So login to Codio, open "project04", and then open a terminal window. Then do the following:

```
cd program
dotnet build
dotnet run
```

If you are working on another platform, you'll want to download the provided files from the course web page. Then you'll probably need to place the input files --- divvy1.csv and divvy2.csv --- where .NET is looking for them. This would be in the sub-folder **bin/Debug/netcoreapp2.2** (note that the digits might be 2.1 or 2.0 or 1.1 depending on what version of .NET core you have installed).

Requirements

No imperative programming. In particular, this means no mutable variables, no loops, and no data structures other than F# lists. Use recursion and higher-order approaches only; you must also work with the input format as given, i.e. a list of lists.

A few F# hints / gotchas…

When computing the average or percentage, you'll need to convert your integer values to real numbers. Use the **float** function: (float integer_value). When a percentage is output, the output contains "%". Since % is a special formatting character of the printf and printfn functions, you need to escape "%" in order to output this character: to escape, simply put two in a row, i.e. printf "%%". Need to print a blank line? Use printfn "". Want to print only 2 decimal places? printfn "%.2f"

When outputting the *'s for the Ride Start Time histogram, it's a little tricky to do recursively because the printf function doesn't return a value --- it's a void function. Here's a recursive function that prints n *'s:

```
let rec printstars n =
  match n with
  | 0 -> ()
  | 1 -> printf "*"
  | _ -> printf "*"
         printstars (n-1)
```

The return value ( ) for the base case of 0 denotes no return value, i.e. void ("unit" in F#).

Electronic Submission and Grading

Grading will be based on the correctness of your console application. We are not concerned with efficiency at this point, only correctness. Note that we will test your app against other input files with the same format.

When you are ready to submit your program for grading, login to Gradescope and upload your F# source file --- this must be named "Program.fs" (you may also submit a .zip containing "Program.fs" or "program/Program.fs"). You have unlimited submissions, and Gradescope keeps a complete history of all submissions you have made. By default, Gradescope records the score of your last submission, but if that score is lower, you can click on "Submission history", select an earlier score, and click "Activate" to select it. The activated submission will be the score that gets recorded, and the submission we grade. If you submit on-time and late, we'll grade the last submission (the late one) unless you activate an earlier submission.

The grade reported by Gradescope will be a tentative one. After the due date, submissions will be manually reviewed to ensure project requirements have been met. Failure to meet a requirement --- e.g. use of mutable variables or loops --- will trigger a large deduction.

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 10%. After 24 hours, no submissions will be accepted.

Unless stated otherwise, all work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your "iClicker teammates"), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .