# Project #02: Employee Payroll Analysis

**Complete By:** Monday March 16th @ noon
**Assignment:** F# program to perform payroll analysis
**Policy:** Individual work only, late work is *not* accepted
**Submission:** via GradeScope

## Employee Payroll

Part of running a business is keeping track of the books, having a record of sales, costs, profits, employees and goods. This data can be stored in a variety of formats, one of which is the humble csv file. Having the data stored is important to keep track of the assets of the company, but has additional use beyond just record keeping. The field of data analysis has widespread popularity in the world today, and can be accomplished in a variety of methods.

This analysis generally does not involve changing the initial data, and so a language without side effects and with a mathematical basis will enable us to cleanly produce an analysis and be confident in the correctness of the produced results.

The various departments employed by the City of Chicago have their payrolls part of the public record. This data is available as part of Chicago's open data portal.

The goal in this assignment is to input the payroll information for a number of employees and perform some analysis of the data: # of employees, the % of salaried and hourly workers, the average salary, and each of these statistics filtered by condition such as by department, etc. Here's a screenshot of the program analyzing the **payroll_01.csv** data file

What follows is a list of functions and descriptions of their behavior you may find useful in producing an analysis of this data.

**1. isSalary employee, isHourly employee, getSalary employee and getDeparment employee:**
The functions isSalary, isHourly , getSalary and getDepartment access specific fields of the employee tuple and return either a boolean (in the case of the is functions) or a value (in the case of the get functions).

**2. calcSalary employee:**
The function calcSalary calculates the annual salary, either by returning the directly recorded annual salary or calculating it by finding the average weekly salary by multiplying the hours per week by hourly wage, and then multiplying that by 52 to get the average annual salary for that hourly worker.

**3. getNumberOfEmployees AllData**
The function getNumberOfEmployees returns the number of employees in the data set.

**4. getNumberOfSalariedEmployees AllData**
The function getNumberOfSalariedEmployees returns the number of employees who have an annual salary in the data set.

**5. getNumberOfHourlyEmployees AllData**
The function getNumberOfSalariedEmployees returns the number of employees who are paid hourly in the data set.

**6. findHighestPaidEmployee AllData**
The function findHighestPaidEmployee returns the name and salary of the highest paid employee. Use the computed salary for hourly employees.

**7. findHighestPaidEmployeeInDept AllData DeptName**
The function findHighestPaidEmployee returns the salary of the highest paid employee within a specific department. Use the computed salary for hourly employees.

**8. getAverageSalary AllData**
The function getAverageSalary calculates the average of the computed salary field.

**9. getAverageSalaryInDept AllData DeptName**
The function getAverageSalary calculates the average of the computed salary field for a specific department.

The function getUniqueDepartmentNames has been provided for you, searching through the data set to generate a list of department names.

**10. howManyEmployeesInEachDepartment AllData DeptNames**
This function computes the number of employees in every department. This function should return a list of tuples, pairs between the department name and number of employees.

**11. findTotalSalaryByDepartment AllData DeptNames**
This function computes the overall annual salary budget for every department.
The calculated salary should include the average annual salary for hourly employees.
This function should return a list of tuples, pairs between the department name and total annual salary.

**12. findHighestPaidDept AllData DeptNames**
This function returns the name and total annual salary
of the department with the largest overall annual salary paid to employees in that department.
The calculated salary should include the average annual salary for hourly employees.
This function should return a single tuple, containing the department name and total annual salary.

**13. printOneHistogram label value amountPerStar**
The function printOneHistogram implements generating a histogram of number of items in a particular group, printing out a number of stars equal to value/amountPerStar after a right shifted standard length label field to line up the graph.

**14. withinSalaryRange lower upper L**
The function withinSalaryRange returns the number of employees whose calculated salary
is greater than the lower bound and less than or equal to the upper bound.

The function printOneHistogram is used to generate two histograms.
In one, a histogram used to output the number of employees in each department
The department names form the labels and the number of employees determine the number of stars.

In the second, employees are grouped by salary range.
The range description forms the label, and the number of employees with a salary in that range determines the number of stars.

The salary ranges used in the output of this program are

```
0        < salary <= 60000
60000    < salary <= 80000
80000    < salary <= 100000
100000 < salary <= 120000
salary > = 120000
```

The program starts by inputting the filename, and a specific department name to analyze.
The program proceeds to output statistics analyzed from the data set, using the functions listed above.
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.

You can solve recursively, or using a higher-order approach (I'd recommend using higher order functions where possible). You cannot use loops, no mutable variables. You should add additional functions as you see fit and may use a different data format in your program if you wish, though you should still only be using lists and tuples.

## File format

The input file consists of at most 10,000 employees. 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 employee. Here's an example:

```
JORDAN M,HARTFIELD,POLICE OFFICER,POLICE,Salary,,76266,
JOHN R,KEATING,FOREMAN OF ELECTRICAL MECHANICS,GENERAL
SERVICES,Hourly,40,,52.35
CHRISTOPHER H,MAROSI,SENIOR PERFORMANCE ANALYST,POLICE,Salary,,72120,
...
```

Each line contains exactly 8 values, in this order:

- First Name: string, First Name of employee, often includes middle initial
- Last Name: string, Last Name of employee
- Occupation: string, title of position held
- Department: string, name of the department employing this person
- Wage Type: string, Salary or Hourly
- Hours Per Week: integer, Number of hours worked on average, generally 20 or 40, not applicable to salaried workers
- Salary: integer, Annual Salary, not applicable to hourly workers
- Hourly Wage: double, wage per hour, only applicable to hourly workers, not applicable to salaried

For testing purposes we are providing two input files: "**payroll_01.csv**" and "**payroll_02.csv**". The first is small so you can inspect the results and compare them with your computations by hand. The second is somewhat larger so you can test whether your solution scales. These files are in the box folder under Projects/Project 02.

## 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 the Box page under Projects/Project 02. Here's the main program as given:

```
//
// F# program to analyze employee salary data.
//
// << YOUR NAME HERE >>
// U. of Illinois, Chicago
// CS 341, Spring 2020
// Project #02
//

module project02

//
// doubleOrNothing
//
// Given a string containing a double numeric value
// or being an empty string
// returns the double equivalent, with the empty string
// treated as the value 0.0
//
let doubleOrNothing s =
    match s with
    | "" -> 0.0
    | x -> double x
```

```
//
// ParseCSVLine and ParseCSVDatabase
//
// Given a sequence of strings representing payroll data,
// parses the strings and returns a list of tuples.  Each
// sub-list denotes one employee.  Example:
//
//[ ("JESSE A", "ACOSTA", "POLICE OFFICER", "POLICE", "Salary", 0.0, 93354.0, 0.0); ... ]
//
// The values are first name, last name, occupation,
// department, salary type, hours per week, annual salary,
// and hourly wage.
// Depending on the salary type,
// either hours per week and hourly wage
// or annual salary will be filled in with 0.0,
// since the field is empty in the csv
//
let ParseCSVLine (line:string) =
    let tokens = line.Split(',')
    let listOfValues = Array.toList tokens
    let fName::lName::Occupation::Department::SalaryType
        ::HoursPerWeek::AnnualSalary::HourlyWage::[] = listOfValues
    (fName,lName,Occupation,Department,SalaryType,
      (doubleOrNothing HoursPerWeek),
      (doubleOrNothing AnnualSalary),
      (doubleOrNothing HourlyWage)
    )

let rec ParseCSVDatabase lines =
    let employees = Seq.map ParseCSVLine lines
    Seq.toList employees

[<EntryPoint>]
let main argv =
  //
  // input file name, then input divvy ride data and build
  // a list of lists:
  //
  printf "Enter name of the csv file containing employee data: "

  let filename = System.Console.ReadLine()
  let contents = System.IO.File.ReadLines(filename)
  let data = ParseCSVDatabase contents

  printfn "This is the data you have loaded."
  List.iter (printfn "%A") data

  0
```

Notice that the **main** function inputs the filename, then opens the file and inputs the data. Each line of the file --- one employee --- is turned into a list of 8 items as described earlier. The resulting data structure is a list of

tuples. I have included an output so you can see the format of the input after parsing. Here's what you'll see working with the smaller input file **payroll_01.csv** (at least this is the start of the output you'll see):

```
Enter name of the csv file containing employee data: payroll_01.csv
This is the data you have loaded.
("JESSE A", "ACOSTA", "POLICE OFFICER", "POLICE", "Salary", 0.0, 93354.0, 0.0)
("AKISHE P", "BROWN", "POLICE OFFICER", "POLICE", "Salary", 0.0, 87006.0, 0.0)
("ANTHONY", "CRONIN", "FIREFIGHTER", "FIRE", "Salary", 0.0, 90024.0, 0.0)
("CONNIE D", "FLORES", "POLICE OFFICER", "POLICE", "Salary", 0.0, 84054.0, 0.0)
```

You are free to use whatever F# programming environment you prefer; submissions will be collected using Gradescope. Then you'll probably need to place the input files --- payroll_01.csv and payroll_02.csv --- where .NET is looking for them. This may be in the sub-folder **bin/Debug** and possibly a folder deeper such as **/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). Also, depending on your operating system, you may need to change the line endings of the file from Windows to Mac or Linux.

## Requirements

No imperative programming. In particular, this means no mutable variables, no loops, and no data structures other than F# lists and tuples. We haven't talked about classes, feel free to experiment with them after the project is done but limit yourself to lists and tuples to practice for the exam. Use recursion and higher-order approaches only.

## A few F# hints / gotchas…

When computing the average or percentage, you'll need to convert your integer values to real numbers. Use the **double** function: (double 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 "".

When outputting the *'s for the 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"). 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.

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 *not* accepted. Please submit before the noon (not 5 pm, not midnight) deadline to receive credit for the assignment.

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 .