Lab 1: Why Databases?
Part I

Due date: Wednesday, September 28, beginning of lab period.

Lab Assignment

Assignment Preparation

This is a pair programming lab. You are responsible for finding your teammates for this assignment, and you need to do it fast. I will assign partners for everyone who has not been able to find one.

This lab assignment should occupy about 2 hours of your lab time.

The Task

The full lab assignment consists of two parts. The first part of the assignment is given to you now. The second part will be given to each team, once the team reports completing the first assignment.

You are given a list of students of a local elementary school together with their class assignment. The list is stored in a file `students.txt`. Each line of the file stores information about exactly one student. The format of the line is:

```
StLastName, StFirstName, Grade, Classroom, Bus, GPA, TLastName, TFirstName
```

Here, `StLastName` and `StFirstName` identify the student, `Grade` specifies the grade the student goes to, `Classroom` specifies the classroom where the student studies, and `TLastName` and `TFirstName` identify the student's teacher. `Bus` is the school bus route that the student takes to come to school and `GPA` is the GPA of the student[1]. `Bus`, `Grade` and `Classroom` are integers

[1] Elementary schools these days don't do GPAs, but we will suspend disbelief for the purposes of this assignment.

(Kindergarden is 0; a student who is arriving to school by own means is 0), while all other fields are strings.

The `students.txt` file is available on the class web page, `http://www.csc.calpoly.edu/~dekhtyar/365-Fall2016/labs/lab01.html`. Download it at the beginning of your work.

Here is a sample line from the file:

```
DROP, SHERMAN, 0, 104, 51, 2.65, NIBLER,  JERLENE
```

Not surprisingly, the line is to be read: "Sherman Drop, who takes bus route 51, is a kindergarden student assigned to the class of Mrs. Jerlene Nibler in the classroom 104. He has a GPA of 2.65."

Your goal is to write a program, that searches the `students.txt` file and outputs the results of the search. The following searches have to be implemented:

- Given a student's last name, find the student's grade, classroom and teacher (if there is more than one student with the same last name, find this information for all students);
- Given a student's last name, find the bus route the student takes (if there is more than one student with the same last name, find this information for all students);
- Given a teacher, find the list of students in his/her class;
- Given a bus route, find all students who take it;
- Find all students at a specified grade level.
- For a given grade level find the average GPA of the students.
- In a given grade level find the student with the highest (lowest) GPA.

Formal Specs

You are to write a program called `schoolsearch`, which implements the requested functionality. The program can be written in any programming language you are comfortable with (Java, C, C++, Perl, Python etc...). The program must satisfy the following requirements.

R1. Either interpreted or compiled version of the program shall run from the command line on our lab machines.

R2. The program shall be invoked without command-line parameters. Upon start, the program shall provide the user with a visible prompt, read search instructions entered by the user, print the result, and wait for the next user instruction until the termination command is recieved.

R3. The following language of search instructions shall be implemented.

- `S[tudent]: <lastname> [B[us]]`
- `T[eacher]: <lastname>`
- `B[us]: <number>`
- `G[rade]: <number> [[H[igh]]|[L[ow]]]`
- `A[verage]: <number>`
- `I[nfo]`
- `Q[uit]`

Notes: For simplicity all instructions are case sensitive. You are welcome but do not have to make them insensitive of the case. In the specs above, square brackets (`[]`) indicate optional parts (e.g., the `Student` instruction can be abbreviated to `S`), while items in angle brackets (`<>`) are the values provided by the user.

R4. `S[tudent]: <lastname>`

When this instruction is issued, your program shall perform the following:

- search the contents of the `students.txt` file for the entry (or entries) for students with the given last name.
- For each entry found, print the last name, first name, grade and classroom assignment for each student found and the name of their teacher (last and first name).

R5. `S[tudent]: <lastname>` B[us]

When the `S[tudent]` instruction is issued with the `B[us]` option, your program shall perform the following:

- search the contents of the `students.txt` file for the entry (or entries) for students with the given last name.
- For each entry found, print the last name, first name and the bus route the student takes.

R6. `T[eacher]: <lastname>`

When this instruction is issued, your program shall perform the following:

- Search the contents of the `students.txt` file for the entries where the last name of the teacher matches the name provided in the instruction.
- For each entry found, print the last and the first name of the student.

R7. `G[rade]: <Number>`

When this instruction is issued your program shall perform the following actions:

- Search the contents of the `students.txt` file for the entries where the student's grade matches the number provided in the instruction.
- For each entry, output the name (last and first) of the student.

R8. `B[us]: <Number>`

When this instruction is issued your program shall perform the following actions:

- Search the contents of the `students.txt` file for the entries where the bus route number matches the number provided in the instruction.
- For each such entry, output the first and the last name of the student and their grade and classroom.

R9. `G[rade]: <Number> H[igh]` or
`G[rade]: <Number> L[ow]`

When this instruction is issued your program shall perform the following actions:

- Search the contents of the `students.txt` file for the entries where the student's grade matches the number provided in the instruction.
- If the `H[igh]` keyword is used in the command, find the entry in the `students.txt` file for the given grade with the *highest* GPA. Report the contents of this entry (name of the student, GPA, teacher, bus route).
- If the `L[ow]` keyword is used in the command, find the entry in the `students.txt` file for the given grade with the *lowest* GPA. Report the contents of this entry (name of the student, GPA, teacher, bus route).

R10. `A[verage]: <Number>`

When this instruction is issued your program shall perform the following actions:

- Search the contents of the `students.txt` file for the entries where the student's grade matches the number provided in the instruction.
- Compute the average GPA score for the entries found. Output the grade level (the number provided in command) and the average GPA score computed.

R11. `I[nfo]`

When this instruction is issued your program shall perform the following actions:

- For each grade (from 0 to 6) compute the total number of students in it.
- Report the number of students in each grade in the format

  `<Grade>: <Number of Students>`

  sorted in ascending order by grade.

R12. `Q[uit]`

When this instruction is issued your program shall quit the current session.

R13. The program shall assume that the file `students.txt` is located in the directory from which it is run, and shall attempt to read the information from that file.

E1. Your program can have minimal error-checking. Basically, exceptions or any other error-causing situations must be handled graciously (without core dumps or runtime error messages), but the program does not need to attempt to recover from most situations. If the `students.txt` is not available, or has the wrong format - exit the program. If the search instruction has different syntax than what is prescribed in **R3**, go back to the prompt.

Implementation notes

The majority of implementation details is left up to individual teams. Depending on the language you choose, different facilities may be available to you. Simple scripting languages make parsing easier and have some convenient data structures (e.g., associative arrays) which may be helpful. Compiled programming languages provide access to large libraries of powerful data structures and programs may work faster, but parsing is not as convenient.

You are encouraged to consult me about your solutions during the lab time (and in other times, via email, during office hours, etc.) about the design and implementation issues.

Tracing, Testing, and Deliverables

Data. The `students.txt` file is available on the class web page, `http://www.csc.calpoly.edu/~dekhtyar/365-Fall2016/labs/lab01.html`. Download it at the beginning of your work, and use it for testing.

Tracing. For this assignment, your source code must incorporate *traceability information.* That is, for each component of your program, you must indicate *in a well-structured comment* which requirements from the specifications above (**R1**—**R13**, and **E1**) this component implements (i.e., *traces to*).

For example, if implementing in Java, a method `computeStudentInfo(..)` can be prefaced with a tracing comment as follows.

```
// Traceability: implements requirements R3, R4, R5

// ... add other comments here

public static <ReturnType> computStudentInfo(...) {
...
}
```

Testing. Create a test suite for your program. The test suit must cover all requirements and test all possible *correct* behaviors of the program, as well as one or two incorrect behaviors (situations where Requirement **E1** is triggered). (Please note, commands that return no information, e.g., `G: 10` do not trigger Requirement **E1**, they simply return no answers. Such commands should also be a part of your test suit).

Each test case is a single command in the command language of the program. For purposes of submission, the test suite shall be represented as a single text file, which includes for each test case an annotation (rendered as a comment) specifying what it tests, and containing the tracing information to the requirement that is being tested.

For example, a simple test suite consisting for three commands can be rendered as follows:

```
// CSC 365. Spring 2017
// Alex Dekhtyar
// Lab 1-1 test suite

// TC-1
// Tests Requirements R3, R4
// short form command name, existing student
// expected output: HAVIR,BOBBIE,2,108,2.88,HAMER,GAVIN

S: HAVIR

// TC-2
// Tests Requirements R3, R4
// short form command name, non-existing student
// expected output: <empty line>

S: DEKHTYAR

//TC-3
// Tests Requirments R3, R13
```

```
// quit command
// expected output: program terminates

Q
```

Writeup. The write-up is a mandatory part of the lab. I recommend that at the beginning of work, you designate one student to prepare it. The write-up shall contain a brief outline of your team's implementation effort. I strongly recommend building your write-up as you go, rather then after the program has been completed. At the very least, the write-up shall contain the following:

- Your team name/number, list of team members;
- Initial decisions: programming language, environment;
- Notes on selected internal architecture: what data structures to use, for what purposes;
- Task log. For each task to be completed, list the name of the task, the student(s) performing it, start time, end time, total person-hours it took to complete. Choose the granularity wisely. You do not have to document every method or function.
- Notes on testing. When, who, how long, how many bugs found, how long it took to fix them.
- Final notes (anything else you want to share with me about your implementation)

Deliverables. The full list of deliverables is specified below. **PLEASE**, make sure the names of each team member are on each deliverable file (except for program output). Also, **PLEASE** make sure that your files are named **EXACTLY AS SPECIFIED** in this document.

1. Your code. The file name shall be `schoolsearch.ext` where `ext` is the extension for the source code files of the programming language you have selected.
2. Any additional source code files. As needed.
3. Your test suite formatted as specified above. Name the test suite file `tests.txt`.
4. Output of running your program on the commands from **your** test suite. Call this file `tests.out`. You can simply paste the program output into a file, or use the Linux `script` command to record your session with your program.

5. Your writeup. The writeup shall be submitted in the form of a PDF document called `writeup1-1.pdf`.
6. `README`. Submit a `README` file specifying how to compile/run your program. If you want to, you can submit a `Makefile` or any other supplemental files that help compile/run your code.
7. `students.txt`. Place `students.txt` file in the submission directory.

Submission. Place all files you want to submit into one directory. `cd` that directory and either `zip` or `tar` and `gzip` the contents of this directory *while in the directory itself*[2].

Name your archive `lab1-1.zip` or `lab1-1.tar.gz`. When you believe you have satisfied all requirements of the lab, submit your archive using the following `handin command`:

```
$ handin dekhtyar lab01 <FILE>
```

Once the submission is made (in-person during the lab time or office hours, via email during all other times), the second part of the assignment will be made available to you.

Submission. Use `handin` to submit as follows:

```
$ handin dekhtyar lab01 <FILES>
```

Good Luck!

[2] I.e., DO NOT zip the directory from its parent.