CS61B, Fall 2004 **Project #1** **P. N. Hilfinger**

Due: Wednesday, 13 October 2004 at 2400

1 Introduction

This first project involves writing a miniature *relational database management system (DBMS)* that stores *tables* of data, where a table consists of some number of labeled *columns* of information. Our system will include a very simple *query language* for extracting information from these tables. For the purposes of this project, we will deal only with very small databases, and therefore will not consider speed and efficiency at all. For that sort of stuff, you might consider taking CS186 at some point.

As an example, consider the following set of three tables containing information about students, about course numbers and locations, and about grades in these courses. Each table has a name (given above the upper-left corner) and each column of each table has a name (given above the double line).

students

SID	Lastname	Firstname	SemEnter	YearEnter	Major
101	Knowles	Jason	F	2003	EECS
102	Chan	Valerie	S	2003	Math
103	Xavier	Jonathan	S	2004	LSUnd
104	Armstrong	Thomas	F	2003	EECS
105	Brown	Shana	S	2004	EECS
106	Chan	Yangfan	F	2003	LSUnd

schedule

CCN	Num	Dept	Time	Room	Sem	Year
21228	61A	EECS	2-3MWF	1 Pimentel	F	2003
21231	61A	EECS	1-2MWF	1 Pimentel	S	2004
21229	61B	EECS	11-12MWF	155 Dwinelle	F	2003
21232	61B	EECS	1-2MWF	2050 VLSB	S	2004
21103	54	Math	1-2MWF	2050 VLSB	F	2003
21105	54	Math	1-2MWF	1 Pimentel	S	2004
21001	1A	English	9-10MWF	2301 Tolman	F	2003
21005	1A	English	230-5TuTh	130 Wheeler	S	2004

enrolled

SID	CCN	Grade
101	21228	B
101	21105	B+
101	21232	A-
101	21001	B
102	21231	A
102	21105	A-
102	21229	A
102	21001	B+
103	21105	B+
103	21005	B+
104	21228	A-
104	21229	B+
104	21105	A-
104	21005	A-
105	21228	A
105	21001	B+
106	21103	A
106	21001	B
106	21231	A

2 Commands.

When you run the database system, it will accept a sequence of commands from the standard input (i.e., normally the terminal), in one of the following formats:

/* *comment* */
    Comments have no effect.

load *table* **;**
    Load data from the file *table*`.db` to create a table named *table*.

insert into *table* **values** *literal* . . . **;**

Add a new row to the given *table* whose values are given by the list of literals (see the description of literals below). There must be exactly one literal for each column of the table, and the table must already exist. This command has no effect if there is already a row in the table with these values.

print *table* **;**

Print all rows of the named *table*, one per line and indented. Separate columns with blanks, and print the columns in the order they were specified when the table was created. See the example below for the format.

select *columns* **from** *table* **where** *conditions* **;**

Find every row of the given *table* (the name of a table) that satisfies the *conditions* (see below), and print the listed *columns* (see example below) for each one (as for the `print` command, above). The "where *conditions*" part may be left off, in which case look at all rows. Print any given set of column values only once.

select *columns* **from** *table1 table2* **where** *conditions* **;**

Find every pair of rows—one from *table1* and one from *table2*—that satisfies the *conditions*, and print the listed *columns* for each such pair. The two tables must be different. Again, The "where. . . " part is optional. Print any given set of column values only once.

name **: select . . . ;**

where "`select ...`" is one of the forms above. Instead of printing the columns found by the "select" part, create a new table, giving it the indicated name. The columns of this table have the names and order of the columns listed by the *columns* part of the "select." If *name* denotes the value of an existing table, replace that table with the new one. Each distinct set of column values gets only one row in the table (no duplicate rows).

quit ;

Exit the program.

exit ;

Synonym for `quit`.

Naming columns. The *columns* specifications in the preceding commands consist of one or more entries of the form *table.name* separated by whitespace, where *table* is the name of one of the tables listed after the '`from`', and *name* is the name of a column of that table. It is an error if a *table* is not one of those listed after `from` or a *name* is not a valid column name for that table. You can leave off the *table* argument, in which case you get the first table listed after `from`. To simplify your life, we'll also require that all the *names* in a given *columns* list be distinct.

Conditions. Each *conditions* clause consists of one or more *tests* separated by the word '`and`'. Each test is of the form $X\ \mathcal{R}\ Y$, where X is a column (as described above), Y is either a column or a *literal*, and $\mathcal{R}$ is a *relation symbol,* one of `=`, `!=`, `<=` `<`, `>=`, or `>`. A literal is just a sequence of valid characters (see Boring Lexical Details below) delimited by single quotes (') on both sides. The relation symbols mean what they do in Java, except that all values are treated as strings (use `compareTo` method on Strings to compare them). Thus you can write things like

```
students.SID = enrolled.SID and students.Lastname >= 'Chan'
```

Boring Lexical Details. Column and table names may consist only of letters, digits, and the underscore character (`_`). The names "from" and "where" are illegal. Values in rows and literals may consist of any printable characters except comma (','), end-of-line characters, and single quotes ('). All the pieces (tokens) of a command must be separated from each other by whitespace (blanks, tabs, ends of lines), except that a column designator (*table.column*) contains no space around the period. You write

```
students.SID = enrolled.SID   not   students.SID=enrolled.SID
load foo ;                    not   load foo;
```

3 Format of .db files

A `.db` file contains first an integer giving the number of columns, followed by that number of strings giving the names of the columns. This is followed by any number of lines containing the values for each row. The items on any line: the number of columns, the column names, and the column values, are separated from each other by commas, which are separated from other data by whitespace (so be careful not to add blanks around values inadvertently.) For example, the 'students' table shown previously would look like this in a file:

```
6,SID,Lastname,Firstname,SemEnter,YearEnter,Major
101,Knowles,Jason,F,2003,EECS
102,Chan,Valerie,S,2003,Math
103,Xavier,Jonathan,S,2004,LSUnd
104,Armstrong,Thomas,F,2003,EECS
105,Brown,Shana,S,2004,EECS
106,Chan,Yangfan,F,2003,LSUnd
```

4 Example

If the information in these tables exists in three files—`students.db`, `schedule.db`, and `enrolled.db`—then a session with our DBMS might look like the following transcript. Characters typed by the user are underlined.

```
DB61B System.  Version 1.0
> load students ;
Loaded students.db
> load enrolled ;
Loaded enrolled.db
> load schedule ;
Loaded schedule.db
> /* What are the names and SIDS of all students whose last name
     is 'Chan'?  */
> select SID Firstname from students
     where students.Lastname = 'Chan' ;
Search results:
  102 Valerie
  106 Yangfan
> /* Who took the course with CCN 21001, and what were their grades?  */
> select students.Firstname students.Lastname enrolled.Grade
```

```
            from students enrolled where enrolled.CCN = '21001'
            and students.SID = enrolled.SID ;
Search results:
  Jason Knowles B
  Shana Brown B+
  Yangfan Chan B
  Valerie Chan B+
> /* Who has taken the course named 61A from EECS? */
> /* First, create a table that contains SIDs and course names */
> enrolled2 :  select enrolled.SID
     from enrolled schedule
     where enrolled.CCN = schedule.CCN and schedule.Dept = 'EECS'
           and schedule.Num = '61A' ;
> /* Print these SIDs */
> print enrolled2 ;
Contents of enrolled2:
  101
  102
  104
  105
  106
> /* Now print the names of the students in this list */
> select students.Firstname students.Lastname from students enrolled2
     where students.SID = enrolled2.SID ;
Search results:
  Jason Knowles
  Valerie Chan
  Thomas Armstrong
  Shana Brown
  Yangfan Chan
> quit ;
```

5 Your Task

The directory `~cs61b/hw/proj1` will contain a few skeleton files that may suggest some structure for this project. Copy them into a fresh directory as a starting point. Please read *General Guidelines for Programming Projects* (see the "homework" page on the class web site). To submit your result, use the command '`submit proj1`'. You will turn in nothing on paper.

Be sure to include tests of your program (yes, that is part of the grade). The makefile we provide has a convenient target for running such tests. Our skeleton directory contains a couple of trivial tests, but *these do not constitute an adequate set of tests!* Make up your tests ahead of time and update your makefile to run them.

The input to your program will come from fallible humans. Therefore, part of the problem is dealing gracefully with errors. When the user makes a syntax error, or mentions a non-existent column, your program should not simply halt and catch fire, but should give some sort of message and then try to get back to a usable state. For syntactic errors (errors in the format of commands) you should skip to the next semicolon (if it hasn't occurred yet) or the end-of-file, whichever comes first. For all errors, you should print a single, separate line starting with the word "error" (in upper or lower case) and followed by any message, and the erroneous command should have no effect on any table. Your program should *not* exit with a cryptic stack trace.

Be sure to include documentation. This is the first project, so the documentation should be very straightforward: a user's manual explaining how to use your program, and a brief internals document describing overall program structure. For the user's manual, *don't* just paraphrase our description; try to "add value" in the form of more helpful examples and perhaps a slower and more gentle treatment.

Our testing of your projects (but not our grading!) will be automated. The testing program will be finicky, so be sure that:

- Your makefile is set up to compile everything on the command `gmake` and to run all your tests on the command `gmake check`. The makefile provided in our skeleton files is already set up to do this. Be sure to keep it up to date if you add additional `.java` files.
- Your main function must be in a class called `db61b`. The skeleton is already set up this way.
- The first line of output of your program identifies the program. It may contain anything.
- Before reading the first command and on reading each subsequent semicolon or comment, your program must print the prompt '`>␣`' (greater than followed by a blank). No cute or obscene phrases, please, just '`>␣`'. If you decide to go the extra mile and output prompts after every end-of-line as well, feel free.
- Output things in exactly the format shown in the example, with no extra adornment.
- Put your error messages on separate lines, starting with the word '`error`' (upper or lower case). The grading program will ignore the text of the message.
- Your program should exit without further output when it encounters a `quit` (or `exit`) command or an end-of-file in the input.
- Your final version must not print any debugging information.
- When printing out the contents of a table, you need not worry about the order of the rows (the autograder will be clever enough to handle any order). However, you *must* print columns in the order specified in the `.db` file or in the *columns* of the `select` command from which the table came.
- The grading program will ignore extra blank lines and extra blanks at the ends of lines, and will treat any run of consecutive blanks as if it were a single blank.

6 Advice

You will find this project challenging enough without helping to make it harder. Much of what might look complicated to you is actually pretty easy, once you take the time to look to see what has already been written for you—specifically what is in the standard Java library of classes. So, before doing any hard work on anything that looks tricky, browse your texts and the documentation.

For reading from files (the `load` command), you'll probably want to use `java.io.FileReader` together with `java.util.Scanner`, which is also good for reading from the standard input. In fact, we've tried to design the syntax to make it particularly easy to parse using `Scanners`. If you find yourself writing lots of complicated junk just to read in and interpret the input, you might back off and ask one of us to look over your approach. The standard Java string type, `java.lang.String`, also contains useful member functions for doing comparisons (including comparisons that ignore case). You will need to choose some way of storing all the rows of a table. You can use arrays or devise a kind of linked list to do so, but you might instead want to jump ahead in the course a bit and take a look at such interfaces as `java.util.List` and `java.util.Map` and at the library classes that implement them. Read the on-line documentation for all of these.

It's important to have *something* working as soon as possible. You'll prevent really serious trouble by doing so. I suggest the following order to getting things working:

1. Write the user documentation.
2. Write test cases (in fact, do this every chance you get). Among other things, writing test cases gets you to understand the specification better. Whenever you find an input that breaks your program, make sure you capture it in a test case.
3. Get the printing of prompts, handling of comments, and the 'quit' and exit' commands, and the end of input to work.
4. Implement the Row class.
5. Implement the parts of the Table class needed to create a new Table, add a Row to it, and print an entire Table.
6. Implement the Database class.
7. Implement `insert` and `load`.
8. The right strategy for `select` (not the most efficient, just the easiest) is to implement the kind that starts with '*name* `:`' first, and then implement the other (printing) kind simply by creating an unnamed table and then printing it rather than storing it away.
9. Implement the kind of `select` that takes a single table and has no conditions.
10. Now get single-table `select` with conditions to work.
11. Finally, work on the two-table variety of `select`.
12. In the above steps, as you decide on implementation strategies and the data representations you will use, write them down in your internal documentation. When you introduce new methods or classes, write the comments first.

You *may* throw all our skeleton code away if you prefer. You are not required to implement `Table`, `Row`, etc. However, we strongly recommend that you *don't* do this if your only reason is that you can't figure out how to do it our way. In that case, pester us with questions until you *do* understand.