Name:

Goal: This assignment asks you to build a small database application for storing information about electronic pictures (e.g., digital photos). The main goal is to gain experience using application programming interfaces (API) to Oracle and PostgreSQL, understanding the common and differing parts. Secondary goals include practice in writing SQL queries and a study of the impact of database design on ease of querying and updating data, and on maintaining database consistency. (Hopefully, this homework will provide a concrete motivation for these topics, which we will study soon.)

The Programming Environment (DBMS libraries): An important part of this assignment is learning the interface between a typical programming environment and the database system. You are free to use a programming language and database interface library of your choice. However, only C (with embedded or dynamic SQL) and Java (with JDBC) are supported. While we will try to help you with other languages and libraries, please note that it is not usually possible to modify software (especially the database system libraries) on the OIT and CSIC machines in the middle of the semester. Figuring out the details of the database system interface and the necessary libraries usually takes people a lot longer than they expect, so please start working on at least this part early! Please use the class newsgroup (and not email) for questions and other discussions.

Database Tables: The application will uses two database tables: Pictures and Makers. The Pictures table has the following columns:

1. PID: a system-generated attribute that uniquely identifies each picture. By system, we mean your program. The type and size of this column is immaterial, as long as it permits at least 10 million unique values. See the paragraph marked Note on Identifiers below for further details.

2. locator: a string that indicates the location of the image file. This string may be a pathname on a local disk (e.g., /home/foo/pics/img321.jpg) or a URL indicating an image on the Web (e.g., http://foo.org/bar/img88.png). You may assume that the length of the locator string is at most 100 characters.

3. ctime: the date and time at which the picture was created.

4. creator: the picture-maker identifier, or PMID, of the person who made the picture. The creator field is a foreign key referencing the PMID column of the Makers table, described below. We will assume that each picture is made by exactly one person.
5. **resx:** the horizontal resolution of the picture, measured in pixels.

6. **resy:** the vertical resolution of the picture, measured in pixels.

7. **lprice:** the list price for this picture, in US dollars.

The **Makers** table has the following columns:

1. **PMID:** a unique identifier (picture-maker identifier) for each picture-maker (person). This column is to the **Makers** table what PID is to the **Pictures** table.

2. **name:** the name of the picture-maker (person). We do not separate names into components such as first and last names, and assume that a name (all parts together) is no longer than 100 characters.

3. **phone:** a contact phone number for the person. You should pick a datatype that allows you to store sufficiently general phone numbers, not just 10-digit ones. For example, your application should be able to manage numbers such as 11 91 22 55 55 12 12 and 1-800-TOP-PICS.

4. **bio:** a short biography of the person. We will restrict biographies to be at most 1000 characters.

5. **essay:** a possibly long essay describing the person, his or her picture-taking style, assorted philosophical remarks, and other miscellany. We will restrict essays to be at most 10000 characters.

**The Application Programs:** As described further in the packaging instructions below, your submission should produce two executable files. The first, called **pdb** (Picture Data Base in Oracle) should implement this application using Oracle to manage the data. The second, called **pdbp**, should be identical in behavior to **pdb**; however, it should use PostgreSQL to manage the data.

You must implement your application program as a Unix command-line program that reads from standard input and writes to standard output. This application must implement the user functions described below. When the work (both internal processing and output to user) for each function is done, your application should write (to standard output) five dashes (-----) followed by a single newline character. We will refer to this string of five dashes followed by a newline as the **function termination string**. The following description also refers to a **separator string**, which consists of the three character sequence space-colon-space. Except the output described in this homework, your program should not produce any extra output such as diagnostic messages. (Before submission, please remember to remove any such messages that you for debugging your code.)

These functions will be invoked from standard input by listing the function name followed by its arguments, one per line. For example, the **connect** function described below takes two arguments and may be invoked as follows (using example values for the arguments):
connect
csc42401
xyzzy

String arguments will be listed verbatim, with no quotes or other demarcation. You may assume that function arguments do not contain any newline characters. Numeric data will be listed in a format 123 or $-123.45$. (That is, integers are listed in common notation and floating point numbers are rounded to two places after the decimal point. There are as many digits before the decimal point as are needed, with no 0-padding. Negative numbers are denoted using a - prefix.) You may assume that all numbers are in the range $[-100,000 \ldots 100,000]$, with at most two digits after the decimal point. Date-time values are in the format YYYY-MM-DD HH:MM:SS. For example, 2001-05-04 14-01-03 denotes three seconds past 2:01pm on the 4th of May, 2001.

The input will contain, in general, several function calls in the above format, listed one after the other. Your program should ignore lines with # (pound sign) as the first character. It should also ignore blank lines, but blank lines separating function invocations are not required. Since you know the number of arguments each function takes, there is no need for such separation. (Note that the function termination string is used only for output, not in the input.) Your application should read and process the functions in the order in which they appear in the input and should terminate gracefully (e.g., by closing open database connections) when the end of input is reached. There is no special end-of-input marker. You do not need to provide any error-handling features; your program will only be tested on valid input.

**Functions:** The functions that your program should implement are described below. Note that the descriptions use a conventional functional notation of the form $f(a, b)$, but the input is presented in the form described above.

**connect($u, p$):** This function will be the first one invoked in any test run, and it will be invoked exactly once per run. In response, your application should perform all necessary initialization and connect to the database server as user $u$ using password $p$. (Strictly speaking, your program need not perform any of these actions, since its observable behavior for this function does not depend on them. However, it is probably a good idea.)

We will test your program using a temporary account $u$ that is *not* your class account. You may assume that the database for account $u$ initially contains no user tables. Make sure you do not assume anything specific to your own class account. For example, you cannot rely on any initialization you have in your .login or .tcshrc files, since these files will not be the same for the test account. Please be sure to understand the implications of this requirement. Creating code that can be easily run by someone else is an important part of this homework. For testing, you should use your own account name and password in place of $u$ and $p$. (You may wish to test your submission by temporarily replacing your customized account files, if any, with the default ones that came with your account.)
createTables(): This function should result in the creation of the Pictures and Makers tables described above. This function will be called before any of the functions below. It does not return any results.

destroyTables(): This function should result in the Pictures and Makers tables being destroyed. The database should now be in its initial pristine state (with no user tables). You may assume that after this function is called, a call to createTable will precede a call to any of the functions described below. This function does not return any results.

addPicture($l, $t, $c, $x, $y, $p$): When this function is invoked, your application should add a tuple ($l, $t, $c, $x, $y, $p$) to the Pictures table, where the attribute values are listed in the order the attributes were introduced earlier. However, duplicates should be avoided: If a tuple ($i, $l, $t, $c, $x, $y, $p$) already exists in the Picture table (for some value $i$ of the PID attribute), no new tuple should be inserted (and no error should be flagged). For each new tuple thus inserted, your program should automatically generate a value $i$ for the PID attribute. This function does not return any results.

addMaker($n, $p, $b, $e$): This function is to the Makers table what addPicture is to the Pictures table. Duplicates should be avoided as described above, with PMID assuming the role of PID.

searchPicturesByLocator($l$): This function should search for pictures with locator values that include the substring $l$. This search, and all searches on string-valued attributes, should be case-insensitive unless specified otherwise. The matching picture records should be printed one per line, sorted in ascending lexicographic order of locators. On each line, the locator should be followed by the separator string (described earlier), in turn followed by the PID of the picture. Output lines here and elsewhere should be terminated by a single newline character.

detailPicture($p$): This function should print all the information for the picture identified by the PID $p$ (exact, case-sensitive string match) on a single line. If there is no record with PID $p$, no output should be produced (and this condition is not an error). The output (if nonempty) should present the attributes of a picture record in the order they were described above.

For this and other functions, attribute values and other items printed on an output line should be separated using the separator string. Strings should be printed literally (with no quotes, padding, or other artifacts). Prices should be printed in the form $123.45. Integers, reals, and dates should be printed in the format used for the input.

detailMaker($m$): This function is to Makers what detailPicture is to Pictures, with records identified using the given PMID $m$. 

4
Note on Identifiers (PIIDs and PMIDs): PID's and PMIDs are identifiers (generated by your program) that uniquely identify a record in the Pictures and Makers tables, respectively. You are responsible for generating and managing these identifiers. Once you have exposed an identifier (by printing it as output), the identifier may be presented as an argument of the detailPicture and detailMaker functions at any point in the future. These identifiers must persist between sessions. For example, if your program exposes a PID 192 during one session (say, in the output of the searchPicturesByLocator function), a detailPicture function call with 192 as the argument must produce details of the picture identified by this PID. Unless this record has been deleted or otherwise modified in the interim, the output of this detailPicture function invocation should be the same as if it had been invoked in the original session. The PMIDs generated by your programs should exhibit similar properties for the Makers records (and corresponding functions). In short, PID's and PMIDs should be persistent and should identify the picture and maker records (respectively) uniquely.

searchPicturesByCTime($t_1, t_2$): This function searches for pictures with ctime from $t_1$ to $t_2$ inclusively. The arguments $t_1$ and $t_2$ are date-time specifications in the format described earlier. For each matching picture, there should be a line of output that lists the matching ctime and the PID of the corresponding picture. The list should be sorted in ascending order of ctime (earliest first).

searchPicturesByCreator($m$): This function searches for pictures with creator $m$ (exact, case-sensitive string match). For each matching picture, an output line consisting of only the PID of that picture is produced. No particular ordering is required.

searchPicturesByResx($x_1, x_2$): This function searches for pictures with resx in $[x_1, x_2]$. Matching records are presented in ascending order of resx values. For each record, there is a line that lists the resx value followed by the PID.

searchPicturesByResy($y_1, y_2$): This function is analogous to searchPicturesByResx, with resy taking the place of resx.

searchPicturesByPrice($p_1, p_2$): This function is also analogous to searchPicturesByResx, with lprice taking the place of resx.

searchPicturesByRes($r_1, r_2$): This function searches for pictures with a resolution in $[r_1, r_2]$, where resolution is defined to be the product of resx and resy. The output lists the resolution and PID of each matching picture, sorted in descending order of resolution.

searchPicturesByCTimeExpr($e$): This function performs wildcard-based matching on picture ctime. The search parameter $e$ is of the form YYYY-MM-DD HH-MM-SS, which is the format we've described earlier for date-time fields. However, in this function zero or more of the components (YYYY, MM, DD, HH, MM, and SS) may be the special wildcard *, denoting a
“don’t care.” Thus, searching for *--01-2001 02--* should list records with a pickup date-time that is between 2:00:00pm and 2:59:59pm (inclusive) on some day in January 2000. The result should list the ctime and PID of each matching picture, ordered by ascending ctime (earliest first).

searchMakersByName(n): This function searches for makers with names matching n (case insensitive substring match). For each matching record, the output has line listing the name followed by the PMID. The output should be sorted in ascending lexicographic order of names.

searchMakersByPhone(p): This function searches for makers with phone numbers matching the given pattern p. The pattern language is as follows: All characters except the period (.), asterisk (*), and backslash (\) are interpreted literally, and match only the corresponding character in the phone number (with case-insensitive matches for letters). A period matches any one character in the phone number, while an asterisk matches zero or more consecutive characters. A backslash is used as an escape character to override the special meaning of the three special characters: A literal period, asterisk, and backslash are represented by \., \*, and \\, respectively. For each maker that has a phone number matching the given pattern p, the output has a line listing the phone number followed by the PMID. The output should be sorted by ascending lexicographic order of phone numbers.

searchMakersByBio(b): This function searches for makers with a bio attribute that matches the given pattern b (case insensitive substring match). For each matching record, the output has a line listing the bio followed by the PMID.

searchMakersByEssay(e): This function is analogous to searchMakersByBio, with e and essay taking the place of b and bio.

listExpensiveMakers(p): The output of this function is composed of one line for each maker whose pictures have an average list price of p or more. For each maker, the corresponding line in the output should list the maker’s name, followed by the average list price for his or her pictures, followed by the number of pictures by that maker. The output should be sorted by descending average price and ascending maker name (secondary sort, lexicographic order).

Packaging You must submit a gzipped tar file containing the source files (not object files or machine code) required to compile and run your program. The file should be named foo.tar.gz (where foo is replaced with something like HendrixJM-1101, as described in PHW01). Unzipping and untarring foo.tar.gz should result in the creation of a single directory (in the current working directory) called phw02. Typing make at the Unix shell prompt in the phw02 directory should result in the complete compilation of your program, producing two executable files (machine code, shell script, Perl script, etc.) called pdo and pdbp, for the Oracle and PostgreSQL implementations, respectively. Obviously, you will need to
include a Makefile in the phw02 directory. You should also include a short README file
describing the files in your submission. This README file is a fallback. If your program
does not work perfectly, we will look at the README file and if it is well written and in-
cludes some special instructions we will try to get your program working by following these
instructions.

Please test very carefully that this unpacking and compilation procedure
works with your submission. Your score will suffer greatly if it does not, or if your
submission contains object files or machine code. (If you use Java, submit the .java files,
not the .class files; your makefile should be designed to produce the .class files. The
make procedure should also result in executable files that run the Oracle and PostgreSQL
versions of the application, perhaps by calling “javaclassname.”) Recap: The sequence of
commands gunzip foo.tar.gz; tar xf foo.tar; cd phw02; make should result in the
final executables pdbo and pdbp. You must make sure you program works with redirecting
input. For example, we may run your program by the command pdbo < datafile, where
datafile is a text file contains the input of the program.

Test Input  You may wish to use this sample input to test your program by replacing
dummyAcct and dummyPassword with your own account name and password. (You should
test your work thoroughly by generating test inputs that exercise all the functions above.)
Note that spaces are significant in string arguments (e.g., passwords, comments) and should
not be ignored or modified. For clarity, the following uses \() to denote the space character.
There is a newline character at the end of each input line.

```plaintext
cconnect
dummyAccount
# This line should be ignored
dummyPassword
ccreateTables
# We may destroy and create the table repeatedly...
destroyTables
ccreateTables

# The above blank line and the one following add should be ignored.
addMaker

Tom Lewis
(301) 405-1234
Ph.D. in Photography (1999)
I feel I was born to take photographs...

searchMakersByName
Lewis

detailMaker
1121
```
Test Output  On the above input, your program should produce the following output. The PID 1121 is arbitrary; your program may produce a different identifier. All the text between the last two function termination lines (------) below is a single line (which has been wrapped below in order to fit on this page).

------
------
------
------

Tom_Lewis::1121

1121::Tom_Lewis::(301)405-1234::Ph.D. in Photography (1999)::I feel I was born to take photographs...

Note:  For the previous input, even when the input ends (Ctrl-D, or EOF), the program should disconnect from the database but not destroy the table, unless explicitly asked. Therefore, any subsequent execution of the program is supposed to start with an existing table, which can be altered, extended, destroyed, searched e.t.c.