EECS-111 Fall 2015 Exercise 2:
A Simple Database

Out: Friday, October 16
Due: Friday, October 23, **noon**

Summary

For this assignment, you'll make a simple database of music albums, like the one used as an example in class, but somewhat bigger. You can use your own albums, other people's albums, or just make them up. Then you will write expressions and procedures that query it for different kinds of information.

As before, we have provided the relevant code from class in the companion Exercise 2.rkt file. Open it in DrRacket, and do **Run**, and then add your code where specified.

Important

Throughout this assignment, as with any assignment, you are encouraged when possible to **reuse procedures** from one problem for solving subsequent problems. Don't just copy the code, however; call the procedure by name. That simultaneously saves you work and makes the code easier to read.

In general, these problems only require **simple** answers. Most have two or three line answers, although the exact number of lines depends on how you put line breaks into your code. But in my solution set, which generally puts line breaks between arguments to a procedure, all problems have solutions of 4 lines or less, except for the last problem, which is 7 lines. That's not to say that if you have an answer that's longer that it's wrong. But it may mean that you're making life **unnecessarily difficult for yourself**. One good way to make life easier on yourself is to reuse code you've already written.

Making the database

Since this is only a toy database, we'll just use a list of **album** objects as we did in class. Each Album object has three fields: the **title**, the **artist** name, and the **genre** (pop, rock, acid-house, country, whatever).

Fill in the definition for database in your Exercise 2.rkt, i.e. type something like this:

```
(define database
  (list (make-album "The White Album"
                    "The Beatles"
                    "Rock")
        (make-album “Collected"
                    "Massive Attack"
                    "Bristol sound")
        (make-album "Idlewild"
                    "Outkast"
                    "Soundtrack")
        ... etc ...))
```

Again, they may not be all your albums, or even actual albums you have; you can even invent fictitious albums. But you need to make sure there are enough albums and that they are varied in the right ways to test your code. For example, one of the procedures you will write is intended to find all the genres in the database. So you probably want to have more than one genre. Similarly, you will write a procedure to find artists who work in multiple genres. Given that you want to make sure there's at least one artist in the database who does work in multiple genres and at least one who does not. That way you can check that the one who does appears in the output but the one who only works in one genre doesn't appear in the output. Feel free to add albums to the database as you go through the assignment if you realize you need more data to properly test your code.

Note that we will be testing your code against our own database.

Querying the database

Okay, now:

1. Write a procedure to find **all the titles** in a database. Call this procedure all-titles. It should take the database as input, so your answer will look something like:

   ```
   (define all-titles
           (λ (db) expression))
   ```

   (you fill in the *expression*). Test your procedure by running (all-titles database) to make sure that it works.

2. Now write one to find **all the artists** in the database. Call this one all-artists. Again, this procedure will take a database as input.

   Important: each artist should appear only once in the output. You can use the remove-duplicates procedure to take a list that has the same items repeated

many times and give you back a new list that only has the one copy of each item. So:

```
(delete-duplicates (list "a" "b" "c" "a"))
```

returns (list "a" "b" "c")

3. Now write a procedure to return all the genres, again with each genre mentioned only once. Call this procedure all-genres. Again, this procedure will take a database as input, so you test it by saying (all-genres database).

Now that you have that under your belt, let's do something a little more complicated:

4. Write a procedure, **artist-albums**, that takes an artist name and database as inputs and outputs all the albums by that artist. That is,

```
(artist-albums "Johnny Cash" database)
```

should return a list with all the albums by Johnny Cash.

Hint: You will need to use filter, and you will need put a procedure expression inside of the call to filter, so your code will look roughly like:

```
(define artist-albums
   (λ (desired-artist db)
     (filter (λ (album)
                 ... fill this in ...)
             db)))
```

Note: you **can't** do this by calling filter with the name of a procedure you've defined separately, as in:

```
(define is-the-right-artist?
  (λ (album) ... some magic code ...))

(define artist-albums
  (λ (desired-artist db)
    (filter  is-this-the-right-artist? db))
```

because the part that says "... *some magic code* ..." would need to use the variables desired-artist and db. But only code inside of artist-albums can access the desired-artist variable. So use a lambda expression inside the call to filter as we did above.

5. Now write a procedure, artist-genres, to return all the genres of a given artist (without duplicates).

6. Now write a procedure, artist-is-versatile?, that takes the name of an artist and the database as inputs and returns true if an artist works in more than one genre. Your solution should call artist-genres.

7. Now write a procedure, versatile-artists, that returns a list of the names of all artists who work in more than one genre. Your solution should call artist-is-versatile?.

8. Write a procedure, artist-album-counts, to count the number of albums by each artist. It should should return a list of lists, where each sublist is the name of an artist followed by the number of albums they have in the database:

```
> (artist-album-counts database)
(list (list "The Beatles" 2) (list "Kronos Quartet" 1) (list "Moondog" 1))
>
```

   Hint:
   a. Start by writing a procedure that computes the number of albums by a single artist.
   b. Now use this procedure to write a procedure that, given the name of an artist, returns the two-element list, the artist's name followed by their album count.
   c. Now use that procedure, along with the all-artists procedure, to write artist-album-counts.

9. Now do the same thing, but count the number of albums in each genre. Call the procedure genre-album-counts. (Hint: start by writing a genre-albums procedure).

Advanced section only

10. Write a procedure, most-prolific-artist, to return the artist with the largest number of albums. This is a little awkward to do with what we've given you. But it will help to know that if you say: (apply max *list*), it will return to you the largest element of *list*. So you have a way of getting a list of the counts of all the albums of all the artists, and of finding what the largest count is. Now all you have to do is go back and find the artist that has that number of albums.[1]

---

[1] You can also do this somewhat more efficiently through the clever use of foldl, but it's simpler to do it the way outlined above.

What to turn in

Make sure that you save your file; also Run it to make sure the final version can be executed without errors. If it does, press the Handin button.