Exercise 3

10/13/2023

100/100 Points

Attempt 1

Review Feedback
10/13/2023

Attempt 1 Score:
100/100

View feedback

Unlimited Attempts Allowed
10/15/2023

Details

For this assignment, you'll make a simple collection of information about music albums (we're calling this a library). This library is just a list of albums - `(listof album)` - so anytime you see library think `(listof album)`. Feel free to use your own albums, other people's albums, or just make them up. Finally, you'll write expressions and functions that query it for different kinds of information.

Important

Throughout this assignment, as with any assignment, you are encouraged when possible to reuse functions from one problem to solve subsequent problems. However, **don't just copy the code**. Call the function by name! That simultaneously saves you work and makes the code easier to read. Plus, if you find a bug later, you only have to fix it in one place: the original definition.

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. 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.

Write your functions to work with any input. Don't assume our test data will look like your test data.

Follow these instructions exactly. If we say to remove duplicates, make sure you remove duplicates. Make sure your type signatures exactly match the ones in the assignment as you go.

Exercise 3 Starter Files (https://bain-cs111.github.io/course-files/exercises/exercise_3_template.zip)

Making Your Library

You'll be adding your code to the `exercise_3.rkt` we've provided for you. Each album object has three fields: the `title`, the `artist` name, and the `genre` (e.g. "pop", "rock", "acid-house", "country", whatever). We've provided the album data definition for you at the top of the file:

```
(define-struct album (title artist genre))
```

Fill in the definition for `testing-library-1` in your definitions file, i.e. type something like this:

```
(define testing-library-1
(list (make-album "Blonde" "Frank Ocean" "R&B")
 (make-album "Malibu" "Anderson .Paak" "Hip-Hop")
 (make-album "Emotion" "Carly Rae Jepsen" "Pop")
 ...etc...))
; a library is a (listof album)
; an album is a (make-album string string string)
```

Don't forget to remove the example album `(make-album "title" "artist" "genre")` from the library!

Again, they may not be all your albums or even actual albums you or exist! 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 functions you will write is intended to find all the genres in the library. So you probably want to have more than one genre. Similarly, you will write a function to find artists who work in multiple genres which means you need to make sure there's at least one artist in the library 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 library *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 library, not yours. So in your tests you should test against a couple of different libraries. We've also included a define in the file for you to fill in an additional library.

Question 1: `all-titles`

Write a function to **find all the titles** in a library. Call this function `all-titles`. It should

```
; all-titles : (listof album) -> (listof string)
; Gets all titles of all albums in the library
(define all-titles
   (lambda (lib) … fill this in …))

(check-expect (all-titles (list (make-album "a" "somebody" "rock")
                                (make-album "b" "somebody 2" "country")))
              (list "a" "b"))
```

> **Hint**: You will probably want to use the `map` function for this.

Question 2: `all-artists`

Now write one to find all the artists in the library. Call this one `all-artists`. Again, this function will take a library as input. Its signature is:

```
; all-artists: (listof album) -> (listof string)
```

It's important that *each artist should appear only once in the output*. Be sure to write a test case (a `check-expect`) for `all-artists` that verifies this (give it a library where there are two albums by the same artist and verify the artist name only appears once in the output).

You can use the `remove-duplicates` function to take a list that has the same items repeated many times and gives you back a new list that only has the one copy of each item. So if we run:

```
> (remove-duplicates (list "a" "b" "c" "a"))
```

We get the result:

```
(list: "a" "b" "c")
```

Question 3: `all-genres`

Now write a function, `all-genres`, to return all the genres, again with each genre mentioned only once. Again, write at least one test case for it so that you know it works. Its signature is:

```
; all-genres: (listof album) -> (listof string)
```

Question 4: `artist-albums`

Write a function, `artist-albums`, that takes an artist name and library as inputs and outputs all the albums by that artist. That is,

```
(artist-albums "K.Flay" some-library)
```

should return a list with all the albums by K.Flay. Again, write test cases to verify the operation of this function. This should output the `album` objects, not just the titles of the albums!

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

```
;; artist-albums : string, (listof album) -> (listof album)
;; Get all albums in the library by the given artist
(define artist-albums
   (λ (desired-artist lib)
     (filter (λ (album) ...fill this in...)
             lib)))
; your tests here
```

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

```
(define is-the-right-artist?
  (λ (album) ...some magic code...))
(define artist-albums
  (λ (desired-artist lib)
    (filter is-this-the-right-artist? lib))
```

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

Question 5: `artist-genres`

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

Its signature is:

```
; artist-genres: string, (listof album) -> (listof string)
```

Again, you should write at least one test case (i.e. a check-expect), for this and all other problems.

> **Hint**: Just call `artist-albums` to find the albums by the artist. Don't rewrite code to do something you've already written code for!

Question 6: `artist-is-versatile?`

Now write a function, `artist-is-versatile?`, that takes the name of an artist and the library as inputs and returns `true` if an artist has albums in more than one genre. Its signature is:

```
; artist-is-versatile?: string, (listof album) -> boolean
```

Your solution should call `artist-genres`.

Question 7: `versatile-artists`

Now write a function, `versatile-artists`, that returns a list of the names of all artists who work in more than one genre. Its signature is:

```
; versatile-artists: (listof album) -> (listof string)
```

Your solution should call `artist-is-versatile?`.

Question 8: `artist-album-counts`

Write a function, `artist-album-counts`, to count the number of albums by each artist. It 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 library. Its signature is:

```
; artist-album-counts:
;    (listof album) -> (listof (listof string number))
```

For example, assuming your library had two Kanye West albums, 1 Hayley Kiyoko album, and 1 Lido album (and nothing else), you'd get:

```
                    (list "Hayley Kiyoko" 1)
                    (list "Lido" 1)))
```

Hints:

- Start by writing a function that computes the number of albums by a single artist.

```
; artist-album-count: string, (listof album) -> number
```

- Now use this function to write a function that, given the name of an artist, returns the two-element list, the artist's name followed by their album count.

```
; artist-album-count-list:
;    string, (listof album) -> (listof string number)
```

- Now use that function, along with the `all-artists` function, to write `artist-album-counts`.

Question 9: `genre-album-counts`

Now do the same thing, but count the number of albums in each genre, rather than the number by each artist. Call the function `genre-album-counts`. Its signature is:

```
; genre-album-counts: (listof album) -> (listof (list string number))
```

Hint: start by writing a `genre-albums` function, then a `genre-album-count` function, etc.

Common Problems

1. Make sure that all of your functions match the provided signatures and that the function is named exactly as specified (and takes the inputs in the order specified).
2. Make sure you follow instructions completely. For example, make sure that artist-genres calls `remove-duplicates`.
3. Don't include any references to `testing-library-1` in your function definitions. This test library **should only be used for your tests (i.e. `check-expects`)**!.
4. Remember that strings are case sensitive. So "Rock" and "rock" are different strings, and hence `(string=? "Rock" "rock")` is false.

Double Checking your Work

Before turning your assignment in, **run the file one last time** to make sure that it runs properly and doesn't generate any exceptions, and all the tests pass. Make sure you've also spent some time writing your OWN `check-expect` calls to test your code. Remember, we will use our OWN ALBUMS as test cases on your functions. That means nothing in your functions can depend on the test albums we gave you (e.g. check to make sure you actually use your inputs; check to make sure it's actually processing those inputs; etc.).

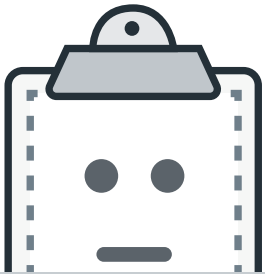
Submitting to Canvas

Questions about submitting to Canvas? About DrRacket features? Checkout our FAQ pages:

- **DrRacket Cheat Sheet (https://canvas.northwestern.edu/courses/201068/pages/drracket-cheatsheet)**
- **Checking for Runtime and Syntax Errors (https://canvas.northwestern.edu/courses/201068/pages/checking-for-syntax-or-runtime-errors-in-racket)**
- **Submission and Grading FAQ (https://canvas.northwestern.edu/courses/201068/pages/exercise-submission-+-grading-faq)**
- **Autograder FAQ (https://canvas.northwestern.edu/courses/201068/pages/autograder-and-submission-faq)**

Remember, **close DrRacket before you submit** to ensure that you've saved your latest work. The Automated Type Checker (ConnorBot) will post its results of its type checks and the built-in `check-expect`s every 30 minutes. If you get an unexpected failure, it may be because you submitted the wrong file.

Late Penalty Waiver



(https://canvas.northwestern.edu/courses/201068/modules/items/2831599)

(https://canvas.northwestern.edu/courses/2

Preview Unavailable

exercise_3.rkt

Download

(https://canvas.northwestern.edu/files/17461594/download?download_frd=1&verifier=gjj2BGgqsUdxM2RWfjQMrDOlpP3WU6zonBp5ri84)

(https://canvas.northwestern.edu/courses/201068/modules/items/2831599)

(https://canvas.northwestern.edu/courses/2