Assignment 1 - Racket Query Language

General Assignment Instructions

- Assignments may be done in pairs.
- You may not import any Racket libraries, unless explicitly told to.
- You may not use mutation or iterative constructions (e.g., for/*)
- You may write helper functions freely; in fact, you are encouraged to do so to keep your code easy to understand.
- Your grade will be determined by a combination of automated tests, and TA grading your work for design, use of higher-order functions, and testing.
- Submit early and often! MarkUs is rather slow close to the due date. It is your responsibility to make sure your work is submitted on time.

We are providing two starter files (links on the assignments web page):

database.rkt

database_tests.rkt

Introduction

As you probably know, the vast majority of application software in use today stores data in some sort of database. One particularly common database structure is a relational database, which organizes data in several tables, and supports operations like defining table structure, inserting and removing data in rows, and performing queries to the data. You may have heard of `SQL`, a commonly used language for manipulating and querying relational databases. In this assignment, you'll implement a subset of the query language portion of `SQL`, in Racket. Not only will you write Racket functions to perform queries on "tables," you'll also use Racket's powerful macro system to mimic the syntax of `SQL`. However, you do not need any prior experience with databases of `SQL` to complete this assignment!

A Database in Racket

Even if you have not heard the term "relational database" before, you are very familiar with this form of data storage. Data in a relational database is stored in tables like this one:

Name	Age	LikesChocolate
David	20	True
Jen	30	True
Paul	100	False

Some terminology:

- Each table is called a relation. A database usually consists of multiple tables.
- Each column of the table is labelled by an attribute (e.g. "Name" or "Age"). The above table has three attributes.

- Each row of the table is called a tuple. The above table has three tuples.

Modern database management systems use very efficient data structures to support fast data manipulation and querying. In this assignment, we'll adopt a more modest goal, and represent a table in Racket as a list of lists in the following format:

- The first list is a list of strings naming the attributes of the relation
- Every other list represents one tuple in the table. For our purposes, the order of the tuples matters!

Of course, the tuples must store values in the order specified by the list of attribute names. The values stored to not need to be strings; any Racket data type is allowed. As an example, we would represent the above database as follows:

```
'(("Name" "Age" "LikesChocolate")
  ("David" 20 #t)
  ("Jen" 30 #t)
  ("Paul" 50 #f))
```

A valid table must satisfy the following properties:

- there must be a list of attributes, which may or may not be empty
- there may or may not be any tuples
- every tuple must have the same length as the list of attributes
- there may be duplicate tuples

For this assignment, you may assume that all input tables to your queries are valid tables (no checking required). However, it is your responsibility to ensure that the output of a query is always a valid table.

Part 0: Semantic aliases

As a warm-up, open up `database.rkt` and complete the functions `attributes`, `tuples`, and `size`. Note that the title of this task contains the term "alias": two of these functions can be exactly identified with existing Racket functions, and the third isn't far off.

Part 1: Functions for the query language

We will now describe the **Racket Query Language (RQL)**, a greatly simplified version of SQL that you will implement for this assignment. Study the following examples carefully: we are introducing both new functionality and new syntax. However, don't worry about how you'll implement this syntax just yet; instead, focus on the functions that will realize this behaviour. Here is the full structure of an RQL query expression:

```
(SELECT <attrs>
 FROM <tables>
 WHERE <condition>
 ORDER BY <order-expr>)
```

We use `<attrs>` to specify the columns we want to see, `<tables>` to specify the table(s) to get the data from, `<condition>` to filter the data, and `<order-expr>` to order the results. The `SELECT` and `FROM` clauses are required for every query, but `WHERE` and `ORDER BY` are optional: zero, one, or both of them may appear

in a query. However, the keywords must follow this order; so `FROM` cannot appear before `SELECT` and `ORDER BY` cannot appear before `WHERE`.
Note that each keyword is paired with an expression; if a keyword appears in a query, then its corresponding expression must appear.
In the sections that follow, we'll go into each of these four parts clauses in more detail. We'll use the following tables as an ongoing example:

```
(define Person
  '(("Name" "Age" "LikesChocolate")
    ("David" 20 #t)
    ("Jen" 30 #t)
    ("Paul" 100 #f)))

(define Teaching
  '(("Name" "Course")
    ("David" "CSC324")
    ("Paul" "CSC108")
    ("David" "CSC343")))
```

Basic selection (`SELECT <attrs> FROM <table>`)

The `SELECT` operation allows you to take a table and return a subset of the columns of that table, corresponding to particular attributes. The result is a new table, including an attribute list.

```
> (SELECT '("Name" "LikesChocolate") FROM Person)
'(("Name" "LikesChocolate")
  ("David" #t)
  ("Jen" #t)
  ("Paul" #f))
```

Note that `SELECT` cannot be a function (why?), but you should be able to write a function that achieves essentially this behaviour. In the returned table, the order of the attributes must be the order specified by `<attrs>` in the query, and the order of the tuples should be the same as the original table (this will change later when `ORDER BY` is used). It is also possible to select all columns from a table by using the literal `*` instead of specifying a list of attributes.

```
> (SELECT * FROM Person)
'(("Name" "Age" "LikesChocolate")
  ("David" 20 #t)
  ("Jen" 30 #t)
  ("Paul" 100 #f))
```

The attributes and tuples in the returned table must appear in the same order as the original table. It is rather redundant to use `SELECT *` on just a single table, but this will change once we start working with multiple tables and filtering out tuples based on certain properties.

Multiple tables (`SELECT <attrs> FROM [<table1> <name1>] [<table2> <name2>] ...`)

Any realistic use of a relational database stores data in more than one table, and so we often want to make queries across multiple tables. For example, we could use our two tables `Person` and `Teaching` to answer questions like "which courses are taught by someone who likes beer?" RQL allows multiple tables to be listed after the `FROM`, as long as each is paired with a name (string). These table names are used to resolve attribute name collisions; you may assume that they will always be distinct. When we make a `(SELECT * FROM [<table1> <name1>] [<table2> <name2>] ...)` query on multiple tables, the resulting table has the following properties:

- Its attributes are the attributes in each table, in the order the tables are listed. If the same attribute appears in more than one table, each occurrence of that attribute is renamed to the form `"<table-name>.<attr-name>"`, where `<table-name>` is the corresponding table name specified in the `FROM` expression. (Do NOT assume that `<attr-name>` has no periods.)
- Its tuples are every possible combination of tuples from the individual tables. For two tables, the combinations appear in the following order: the first tuple of the first table is paired with each tuple from the second table, then the second tuple of the first table is paired with each tuple from the second, etc. For more than two tables, the construction of tuples is recursive: first tuple of the first table is paired with every tuple in the combination of the remaining tables, and then the second tuple of the first table, etc. (Note: the same order is used by the function `cartesian-product` in Exercise 2.)

Note: you can assume that the `[<table> <name>]` form is only used when joining two or more tables; if a query is made on just a single table, a name cannot be provided.
Of course, it is also possible to select just a subset of the attributes from the joined table as well.

```
> (SELECT '("P.Name" "Age" "T.Name" "Course") FROM [Person "P"] [Teaching "T"])
'(("P.Name" "Age" "T.Name" "Course")
  ("David" 20 "David" "CSC324")
  ("David" 20 "Paul" "CSC108")
  ("David" 20 "David" "CSC343")
  ("Jen" 30 "David" "CSC324")
  ("Jen" 30 "Paul" "CSC108")
  ("Jen" 30 "Daivd" "CSC343")
  ("Paul" 100 "David" "CSC324")
  ("Paul" 100 "Paul" "CSC108")
  ("Paul" 100 "David" "CSC343"))
```

There are plenty of non-sensical tuples produced by this query. Keep reading!

Filtering tuples (... WHERE <pred>)

By adding a `WHERE <pred>` component to a query, we filter the results to only see the ones we care about. Here, `<pred>` is an expression that contains strings referring to attributes in the table. The query returns a table containing the tuples whose values "satisfy" `<pred>` (possibly with only certain attributes selected). Here, a tuple satisfies `<pred>` if and only if when its values are substituted for the corresponding attribute names in `<pred>`, the expression evaluates to `#t`.

```
> (SELECT * FROM Person WHERE (> "Age" 25))
'(("Name" "Age" "LikesChocolate")
  ("Jen" 30 #t)
  ("Paul" 100 #f))
```

Because we represent attribute names as strings, you may worry about ambiguity in expressions: does "Age" refer to an attribute, or just a string literal? For the purposes of this assignment, you can assume that **a string literal in an expression refers to an attribute if and only if it is equal to that attribute name**. So for example, in the query

```
> (SELECT * FROM Person WHERE (equal? "Name" "Joe"))
```

the string `"Name"` refers to an attribute in the `Person` table, and the string `"David"` does not. Here's a query for "which courses are taught by someone who likes beer?"

```
> (SELECT '("Course")
   FROM [Person "P"] [Teaching "T"]
   WHERE (And "LikesChocolate" (equal? "P.Name" "T.Name")))
'(("Course")
```

```
  ("CSC324")
  ("CSC343"))
```

Subtlety I: Not and!

Note that in the starter code and the above query, we used `And` (our own function) instead of the built-in syntactic form `and`. We have provided extra forms for `or` and `if` too; please use these instead of the original Racket forms. This will make your life easier in Part 3 (you can explore why on your own).

Subtlety II: pred is not a function!

Note that to make our syntax match `SQL`'s as closely as possible, we do not give `WHERE` a predicate function like you might have expected. Indeed, if you try to evaluate the expression `(And "LikesBeer" (equal? "P.Name" "T.Name"))`, you'll always get `#f`!

The substitution of values into ¡pred¿ to get a meaningful boolean value for each tuple is probably the hardest part of this assignment. There are a few intermediate functions that you may wish to implement described in the starter code. You are not required to do this, and may choose to use a slightly or very different strategy, but hopefully this gives you some ideas.

ORDER BY (... ORDER BY <ord>)

Sometimes we care about the order in which the query results are presented. In these cases, we can use the `ORDER BY` keyword to sort the resulting tuples. Here, `<ord>` is an expression involving attribute(s), similar to the ¡pred¿ from WHERE, except `<ord>` must evaluate to a number rather than a boolean. Each tuple of the table is sorted in non-increasing order of the value produced by `<ord>`.

Here are two example of sorting, one in which `<ord>` is simply the name of an attribute, and another which involves calling a Racket function in the ordering.

```
> (SELECT '("Name" "LikesBeer")
    FROM Person
    ORDER BY "Age")
'(("Name" "Age" "LikesBeer")
  ("Paul" #f)
  ("Jen" #t)
  ("David" #t))
> (SELECT *
    FROM Person
    ORDER BY (string-length "Name"))
'(("Name" "Age" "LikesBeer")
  ("David" 20 #t)
  ("Paul" 100 #f)
  ("Jen" 30 #t))
```

Ties should be broken based on the order the tuples appeared in the original table. Note that Racket's built-in `sort` function is stable, and you are encouraged to use it. Unlike some implementations of `SQL`, the attributes used in the `<ord>` expression do not necessarily have to appear in the `SELECT` clause.

RQL order of operations

Something that has been implicit so far is the order in which the clauses take effect during the query. This is actually quite important, as varying the order of application can have dramatic impacts on the results of the query.

Consider a generic query with all four components:

```
(SELECT <attrs>
 FROM <tables>
 WHERE <condition>
 ORDER BY <order-expr>)
```

This is evaluated in the following order:

1. If there is more than one table specified after `FROM`, they are all joined together to form a single combined table.
2. The `<condition>` is applied to the combined table to filter out certain tuples. (Note that attributes named in `<condition>` must refer to the attributes of the combined table, not necessarily the original attribute names.) The remaining tuples are ordered according to `<order-expr>` (again referring to the attributes of the combined table). The columns specified by `<attrs>` (again, combined attributes) are returned in a new table.

Checkpoint!

By this point, you should have written functions to accomplish the bulk of the underlying functionality for the four major components of RQL.

Please make sure that you've tested these functions thoroughly! You want to be very confident that your functions work correctly before you add the extra macro syntax later on top of them. Note that none of the provided tests will work at this point, as they rely on the actual RQL syntax you will develop using macros. This means you'll have to write your own tests!

Part 2: completing the tests

Before you move onto actually writing the macros, let's take advantage of the fact that you've started the assignment early, but we haven't yet covered enough in lecture to actually realize RQL in code. This means that you have even more impetus than usual to **write tests before implementation**!

Open the starter code for the tests and read through the tests. Each of the tests contains a valid query, as well as the expected result for that query. Making sure you understand each of the tests is an excellent way to reinforce your work in the previous part.

These tests are just a subset of the ones we will run on your assignment; your task for this part is to complete these tests by covering all of the missed corner cases. What do we mean by corner cases? Here are some ideas to get you started, but it is your job to read this handout carefully to decide what tests are missing!

- Tables can be empty, or just have one tuple
- Tables can have duplicate tuples
- Predicates in the `WHERE` clause may not actually involve any attributes

You must submit your work in the file `database_tests.rkt`.

Part 3: Macros

Take the all of the possible syntactic forms described in **Part 1** and, on paper, *map* them to your functions. For example, suppose you wrote a function select-columns that takes a list of attributes and a table and returns the columns of the table corresponding to the attributes. Then you'll probably want to map the query `(SELECT <attrs> FROM <table>)` to the function call expression `(select-columns <attrs> <table>)`.

Of course, this is just a simple example; you are responsible for handling all sorts of queries, with or without combinations of `WHERE` and `ORDER BY`. You should be able to create these mappings for most functionality on paper first!

After you learn about macros, you will be able to implement your mappings in `database.rkt`, perhaps with some modifications to handle the Racket macro syntax. Take your test suite from `Part 2`, build your macros up slowly, and uncomment tests section by section!

Clarification: Two requirements which you should think about for your macros:

- You should be able to make any query with both an identifier referring to a table, and a literal nested list representing a table (see tests for examples)
- Since queries return tables, it should be possible to *nest* queries (again, see tests).

Expressions with attributes

As we said above, the hardest part of the assignment is converting a condition involving attributes like `(< "Age" 50)` into a function that will take in a tuple and return either `#t` or `#f`.

It turns out that this can be done elegantly using a straightforward recursive macro that traverses the expression tree, somehow using `replace-attr` (see starter code for **Part 1**). Of course, it is your job to figure out exactly how to do this, but here is a skeleton macro to get you started (change as needed):

```
; What should this macro do?
(define-syntax replace
  (syntax-rules ()
    ; The recursive step, when given a compound expression
    [(replace (expr ...) table)
     ???]
    ; The base case, when given just an atom. This is easier!
    [(replace atom table)]))
```

Submitting the assignment

Once you are finished and confident in your work, follow these steps to submit your assignment on MarkUs.

1. Login to MarkUs.
2. Go to the "Assignment 1" page.
3. If you worked in a group, form a group and make sure your partner joins. Remember that this requires the "invitee" to actually login and join the group; otherwise, that person won't receive any credit for the assignment.
4. Submit two files:
   - database.rkt, containing your implementation of RQL
   - database_tests.rkt, containing your tests'
5. In a fresh directory, download your submission and run your tests. Don't proceed until they do!
6. Congratulations, you're done!