Comp 200 Fall 2019 Project 3
Due date: December 2, 2019

- **Warnings**

  - Please read the "Project 3" handout carefully
  - Download your template project3.scm from the website or from the share folder:
    `DriveF@VOL/COURSES/UGRADS/COMP200/SHARE/project3_files/`.
    There is an additional file (also in the same folder) or here: databases.scm
  - Please do the necessary editing on your project3.scm and make sure your file loads without errors by using RUN (DrRacket) command. **Only the files that are error-free will be graded.**
  - Name of the submission file has to be **project3.scm**. If you need to resubmit, the format should be the following: **project3_X.scm** where X is the submission number. For example, if you want to re-submit your homework for a **second time**, the name of file should be **project3_2.scm** . **Files that are named other than that will not be graded.**
  - Please upload your homework to the folder
    `DriveF@VOL/COURSES/UGRADS/COMP200/HOMEWORK/[username]/project3/`.
  - In *DrRacket*, please select the **R5RS** option from the language toolbar.

- **Help** Please send your questions to helpline comp200-group@ku.edu.tr

Purpose

Project 3 focuses on the implementation and use of relational databases. Higher-order procedures, data structures, tagged data, and data abstraction all play a role in the project. In addition to implementing relational database operations, you will also extend the system, and then explore its use with a real dataset. The last part will use life expectancy data (you can find a link to this report on the project web site) from the U.S. Department of Health and Human Services.

Relational Databases

In the lecture, a simple table abstract data type (ADT) was described, where each entry in a table consists of a (key,value) pair. In this project, on the other hand, we consider *relational database tables* that generalize our lecture tables in a number of important ways. First, each entry or "row" in a relational table can store several values (v1,v2,v3,...), each value being in a distinct "column." Second, any column can be considered to hold a "key," depending on the use of the table. For example, one can formulate a "query" on the table to find one or more rows which have a particular value, or satisfy constraints on one or more values. Finally, the tables from lecture depended on *unique* keys; adding another (key, value) pair where the key was already in the table replaced the previous value associated with the key with the new value. In contrast, we will see that relational tables do not require that every key be unique.

Each table in a relational database stores information about a "relationship" between values. A table consists of an unordered list of rows and an ordered list of columns. Each row of the table has a value for each column. Each column has a name and a type of data stored in that column. For example, a table which stores a relation between name and major:

name	major
ben	6
jen	3
amy	12
kim	13
alex	6

This table has five rows and two columns. Though not shown in the table above, the name column has type symbol and the major column has type number. Attempting to add a row that doesn't meet these type constraints will yield an error. This careful type-checking will lessen the chance of database corruption escaping unnoticed.

Operations on Relational Tables

The following operations are specified on relational tables. Some of these operations are defined in `databases.scm`; in other cases, you will complete these procedures as part of this project.

- **make-empty-table** columns

  Creates a table, given a list of column names and types, and returns an empty table. Once created, the columns for a table never change.

- **table-insert!** row-data table

  The row is inserted into the table if the data in the row matches the expected types of the columns for the table. Rows might be inserted anywhere, as the table doesn't specify any default ordering of the rows. Note that the same row of data can be entered more than once in the table; this is unlike the simple tables from lecture where the key had to be unique. The `table-insert!` operation **modifies an existing table** (and returns a pointer to the table as the return value), as opposed to creating a new table with the row inserted. This procedure is provided to you; for the moment, ignore the mechanics of this modification (it will be covered in the lecture on mutation).

- **table-delete!** predicate table

  Rows are deleted from the table if the provided predicate returns true when applied to the row. Similar to insert, this operation **modifies the table** (and returns a pointer to the table as the return value).

- **table-select** predicate table

  A new table is created and returned with the same columns, but containing only those rows for which the provided predicate returns true. A select using the predicate

  ```
  (lambda (row) (= (get 'major row) 6))
  ```

  returns the table:

name	major
ben	6
alex	6

- **table-update!** predicate column updateproc table

  Modifies an existing table (and returns a pointer to the table as the result), changing those rows for which the predicate returns true. A single column is updated, with the new value being computed by another provided procedure. For example,

  ```
  (table-update (lambda (row) (eq? (get 'name row) 'amy))
                'major
                (lambda (row) 6)
                table)
  ```

  modifies the table, setting amy's major to 6.

- **table-order-by** column table

  Creates and returns a new version of the specified table, with rows ordered based on the values of a particular column from the table. Our example table, ordered by major:

name	major
jen	3
ben	6
alex	6
amy	12
kim	13

Many of these operations are similar to the SQL commands of the similar name (e.g. insert, delete, select, update). SQL (Structured Query Language) is used to manipulate almost every relational database on the market.

Database implementation

Read over the code in `database.scm`. Check the DrRacket references manual as needed if there are procedures or special forms being used that you are unfamiliar with.

Types

A type-table contains an association list of type name, and a pair of procedures, the checker and the comparator, for operating on that type.

- the **checker** returns true if a given value is of the specified type
- the **comparator** compares two values of the type, and returns true if the first one is "less than" the second one. Essentially, this provides an ordering over the values of a given type, so that they can be sorted.

Other abstractions

Make sure to familiarize yourself with the column, row and table abstractions. The following annotated picture of the printed representation of the example table may help:

Part 1: Table Implementation

In this part, you will complete the implementation of the table abstraction. For this part, a limited set of example and test cases have been provided in the file `project3.scm`. These cases will help illustrate what is expected for the procedures that you are to implement. You are expected to demonstrate your code running on these test cases, and on additional test cases that you make up yourself.

Problem 1

Using `make-empty-table` and `table-insert!`, create the example table used above. Remember that the order of rows doesn't matter. Include a printout of the `table-display` of the table.

Problem 2

Write `table-insert-all!`. It should take a list of row-data to insert and use `table-insert!` to insert them into the table. Test this on the supplied example (uncomment it in the code file).

Problem 3

Write `table-select`. It should take a predicate procedure and a table, and return a new table with only rows for which the predicate returns true on the original table. Remember that you can use constructors and selectors for tables to get out the pieces, and then build a new table.

Problem 4

Write `table-order-by`. It should return a new table with the rows ordered by the given column. The ordering should be done with the supplied procedure `sort`.

Also remember that several utility procedures are provided in the `databases.scm` file that may be helpful, particularly those related to comparators. In particular, think about how to get the comparator associated with the type of data stored in a given column, and then how to sort the row data of a table based on the values of the associated columns, and finally how to build a new table out of the sorted row data.

Problem 5

Write `table-delete!`. Delete rows from the table where the predicate returns true. You should use the internal procedure `change-table-data!` (which is provided to you in `databases.scm`) to modify the table.

Problem 6

Write `table-update!`. This should take as arguments a predicate to apply to a row, a column name, a procedure to compute a new value for the entry in that column, and a table. For every row which the predicate returns true, use `rowcol-replace` to build a new row replacing the column value with the result of calling the proc on the row. Remember to signal errors if the value doesn't match with the column type.

Part 2: Adding Enumerated Types

Problem 7

Our relational database has an associated set of types, so that type checking on each table entry can be performed. As provided, the table includes the `symbol` and `number` types. Add a string type to the system, and demonstrate it by creating a table with string elements. Verify that your `table-order-by` procedure works with this new type. Why do we need a string type if we have symbols?

Problem 8

In addition to numbers, strings and symbols, we'd like to support a number of enumerated types as database column types. An enumerated type is one where values are drawn from a list of symbols. For example, the enumerated type `days` might allow the values `sunday`, `monday`, `tuesday`, `wednesday`, `thursday`, `friday`, and `saturday`. Assuming that days are ordered in this fashion and that the checker has been defined, the following would work:

```
(day-checker 'monday)    => #t
(day-checker 7)          => #f
```

For a comparator which has been defined such that days appearing earlier in the above list are "less than" days appearing later, the following results would be obtained:

```
(day-comparator 'monday 'tuesday)   => #t (monday is "less than"  tuesday)
(day-comparator 'friday 'sunday)    => #f (sunday is before friday)
```

We'll define an enumerated type as a list of symbols. In order to use enumerated types in our type-table, we need to come up with a checker and a comparator. Instead of writing individual checkers and comparators for each enumerated type, we can write a higher-order procedure, that given the list of symbols that make up the enumerated type, returns an appropriate checker or comparator for the type.

Write `make-enum-checker` and `make-enum-comparator`. Test with days:

```
(define *days*
      '(sunday monday tuesday Wednesday thursday friday saturday))
(define day-checker (make-enum-checker *days*))
(define day-comparator (make-enum-comparator *days*))
```

Add the `day` type to the system, and demonstrate that it works by creating and populating a table with a column of type `day`. Show that you can order your table entries based on this column.

Part 3: Data Abstractions Using Tables

Relational databases are often used as the storage mechanism for other abstractions. Consider the notion of a data abstraction: the user of the abstraction interacts with the data through a number of procedures, and the implementor of the abstraction is free to store the data in any way they choose.

We have previously discussed the notion of *constructors* and *selectors* as being common parts of a data abstraction. To this we add the idea of *mutators*, or procedures which *change* some component of an existing data item. Together, the constructors, selectors, mutators, and the contract that governs the use and behavior of these procedures make up a data abstraction; in this case, the abstraction will be implemented using a relational database.

In order to experiment with this idea, you are going to implement the abstraction for a person. A person has a **name**, **race**, **gender**, and **birthyear**. In the constructor, instead of just making a list out of these values, we're going to store them in a relational table. We will assume that all names are unique (no two people have the same name). Thus, a person can be uniquely identified (their row found in the table) based only on their name.

Note: It is a point of debate about whether race is a legitimate or well defined attribute for individuals, and whether data should be collected or tabulated on the basis of such attributes. This raises an important point: programmers are confronted with ethical questions, and the software systems they build can reinforce and/or change the structure of our society in subtle ways. See "Code and Other Laws of Cyberspace" by Lawrence Lessig.

Problem 9

Before we can build a table for the `person` abstraction, we need to add support for new column types. Add two new enumerated types, one for gender (male, female) and one for race (containing at least black and

white enumerations) and enter them into the type-table. Note that we are including these gender and race enumerations so that we may later use the life-tables data drawn from US Department of HHS publications. You may include any additional race enumerations you would like in the race type, and the ordering on these types can be arbitrary. Show examples of using these additional enumerated types.

Problem 10

Build a table to store this type of information. Pick appropriate types for the columns. Add some people to the table to verify it works.

Problem 11

Complete the `make-person` constructor, so that it adds the person's data to the table you created.

```
define (make-person name race gender birthyear)
   your-code-here
   name)
```

You will note that `make-person` returns only the person's name. Remember that this is sufficient to extract the rest of the information from the table, so that the name becomes the "identifier" or handle for retrieving information about that person later.

Make a couple of people, then check the table to ensure they made it there. You'll want to clean out the "test" people you created for Problem 10; luckily a single `table-delete!` operation will remove all of them.

Problem 12

Now we need some selectors. One of them is very easy:

```
(define (person-name person)
  person)
```

However, the rest are a little harder. Since they all share the common pattern of looking up the name in the table, let's abstract that into the following procedure:

```
(define (lookup-person-row person)
  your-code-here)
```

This procedure should return the row in the table corresponding to the name input. If the caller supplies a bogus name, signal an appropriate error. Once you've written this, the rest of the selectors are also easy:

```
(define (person-race person)
  (get 'race (lookup-person-row person)))

(define (person-gender person)
  (get 'gender (lookup-person-row person)))

(define (person-birthyear person)
  (get 'birthyear (lookup-person-row person)))

(define (person-age person)
  ; returns an approximation to the person's age in years
  (let ((*current-year* 2019))
    (- *current-year* (person-birthyear person))))
```

Demonstrate the selectors working on some examples.

Problem 13

Similar to the above, we want to implement some mutators (that is, some operations that change the data for a given row). The common pattern for each change to the person item is to update the person's row in the table:

```
(define (update-person-row! person colname newvalue)
  your-code-here)
```

Again, once we have this procedure, the individual mutators are easy:

```
(define (set-person-name! person newname)
  (update-person-row! person 'name newname))

(define (set-person-race! person newrace)
  (update-person-row! person 'race newrace))

(define (set-person-gender! person newgender)
  (update-person-row! person 'gender newgender))

(define (set-person-birthyear! person newbirthyear)
  (update-person-row! person 'birthyear newbirthyear))
```

Demonstrate the mutators working on some examples (and show that the changes stick!). One example of interest:

```
(define alyssa (make-person 'alyssa-p-hacker 'black 'female 1978))
(set-person-name! alyssa 'alyssa-p-hacker-bitdiddle)  ; got married!
```

```
(person-name alyssa)
(person-race alyssa)
```

What happens? Why? Comments?

Part 4: Life Expectancy Data

You'll find the published life-expectancy data for the US in the file `databases.scm`. This data lists the life-expectancy of people living in the US by a subset of race, gender, and year. If you're curious, you can look at source of this data in the original Vital Statistics report (Table 19). Load the census data file.

Problem 14

Create and populate a table consistent with the columns and rows in the lifeexpectancy data (i.e. has 10 columns according to the organization of `life-expect-data` in `databases.scm`). Turn in the display of a select statement that shows the life-expectancy in the 1950s.

Problem 15

Unfortunately, the raw data table created in Problem 14 is not quite in the format we would like to support meaningful queries. We'd like to have the original life expectancy data as a table in the form (year race gender expectedlifespan). Create a new table with these columns, and write an expression to populate this new table with data from the old (note that this may require more than one table insertion).

Problem 16

Using the census data table from Problem 15, write a statement that creates a new table which is (1) ordered from lowest to highest life expectancy, for (2) white females, for (3) the years 1950 to 1959. Display the table resulting from this query. Was life expectancy for white women steadily increasing in this decade?

Problem 17

We will interpret the vital statistics data as saying, for example, that on average a person born in 1910 has a life expectancy of 50 years (the "all-all" value in the 1910 row), under the hypothetical condition that they were to be subjected to the prevailing death rates in 1910 for their entire life. The data is a "period life table," in contrast to a "cohort life table," as explained in the introduction to the National Vital Statistics Report for this data.

Write a procedure that, when given a person, computes how many years longer that person might expect to live (based on the period life table data).

```
(define (expected-years person)
  your-code-here)
```

As usual, demonstrate your code running on examples. Aren't you glad you're living now, and not seventy years ago?