# Cmpe 260 Scheme Project

Due Date: May 11, 2014 23:55 PM

> *The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three:*
>
> 1. *Combining several simple ideas into one compound one, and thus all complex ideas are made.*
> 2. *The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations.*
> 3. *The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its general ideas are made.*
>
> – John Locke, *An Essay Concerning Human Understanding* (1690)

## 1 Introduction

In this project, you're going to implement a *very simple* in-memory data store. Actual in-memory databases have a much complicated internal structure to provide efficiency, atomicity, consistency, durability; some of which you're going to see in Cmpe 321 *Introduction to Databases* course. In contrast to complicated architecture and design concerns of database management systems, this project is kept rather simple as an exercise on functional programming.

## 2 Tables

Your in-memory data store has only one fundamental data structure: tables. Although Scheme provides nice ways to construct objects and classes to create such data structures with encapsulation, it's out of scope of this project. Our fundamental data structure for tables is, as you all expect, *lists*. A table is actually a list in the form:

```
'(table-name fields values ...)
; Example:
'(students (name id gpa) (ali 1 3.2) (ayse 2 3.7))
```

Here `car` of a table is its name, `cadr` of a table is names of fields and `cddr` of a table is a list of rows in a table. We assume that table-name is a symbol, fields is a list of symbols, values are a list of lists where each list correspond to a row in table.

## 3 The functions

We indeed have a very simple table structure, but we need some functions to maintain them. Our *functions* are functions in mathematical sense, they ***do not*** create any side effect, change a value, print something to

screen or depend on a global value. This is an important feature of functional programming and it indeed makes unit testing (in a real environment) much easier.

### 3.1 `(table? a-value)`

The `table?` function takes a value of any kind and returns a Boolean value indicating whether it's argument is a table. This function shouldn't fail to run. Some examples:

```
> (table? '(students (name id gpa) (ali 1 3.2) (ayse 2 3.7)))
=> true
> (table? '(students (name id gpa)))
=> true
> (table? '(students))
=> false
> (table? '(students (name id gpa) (ali 1 3.2) (ayse 2 3.7 eggs)))
=> false
> (table? '(students spam (ali 1 3.2) (ayse 2 3.7)))
=> false
> (table? '(students (name id gpa) spam eggs (ayse 2 3.7)))
=> false
> (table? '((some list) (name id gpa) (ali 1 3.2) (ayse 2 3.7)))
=> false
> (table? '(42 (name id gpa) (ali 1 3.2) (ayse 2 3.7)))
=> false
```

Here,

- First two cases are valid tables.
- Third case lacks a list of fields.
- In fourth case, the number of fields and cells in last row mismatch, hence it is *not* a table.
- In fifth case, fields is not a list.
- In sixth case, some rows are not lists.
- In seventh and eighth cases, table names are not symbols.

### 3.2 `(create-table table-name fields)`

This function creates an empty table whose name is `table-name` and fields are given in `fields` list. Here is an example:

```
=> (create-table 'students '(name id gpa))
'(students (name id gpa))
```

From here on, assume that we have following definition in examples:

```
(define student-table
  '(students (name id gpa) (ali 1 3.2) (ayse 2 3.7)))
```

### 3.3 `(get table row field)`

Returns value of `field` in given `row`. Uses `table` to resolve field names. Assuming `field` is a field of `table`. An example:

```
> (get student-table '(ali 1 3.2) 'gpa)
=> 3.2
```

### 3.4 `(alter table row fields-values)`

Alters values given in `fields-values` in given `row`. Uses `table` to resolve field names. Doesn't change original `row`, instead returns a new one. `fields-values` is a list of field-value pairs. An example:

```
> (alter student-table '(ali 1 3.2) '((name ahmet) (gpa 3.3)))
=> '(ahmet 1 3.3)
```

Here we changed values of `name` field to `'ahmet` and `gpa` field to `3.3`.

### 3.5 `(add-row table row)`

Adds a row to a table. Adding a row to end of a table is inefficient (it takes $O(N)$ time where $N$ is number of rows), so we're adding the row *at the beginning*! Here is an example:

```
> (add-row student-table '(asli 3 2.8))
=> '(students (name id gpa) (asli 3 2.8) (ali 1 3.2) (ayse 2 3.7))
```

Note that original table *is not changed*, instead our function returned a new table. As stated above, our functions do not change.

### 3.6 `(add-rows table rows ...)`

Adds many rows to a table. Kind of a bulk addition operator. An example:

```
> (add-rows student-table '(asli 3 2.8) '(ahmet 4 0.8))
=> '(students (name id gpa) (asli 3 2.8) (ahmet 4 0.8) (ali 1 3.2) (ayse 2 3.7))
```

The order which you add the rows in this case may change, use the one that's suitable for you. This is a variadic function, that is it's arity is indefinite. The ways to define such functions and usage of them will be given in the upcoming PS. Hint: you can use `foldr` or `foldl` for defining this function.

### 3.7 `(delete-rows table predicate)`

This function deletes rows which `predicate` returns true. Here `predicate` is a function that takes a table and a row then returns a Boolean value, i.e. it has type signature $table? \times list? \rightarrow boolean?$. Here is an example:

```
> (delete-rows student-table
                (lambda (table row)
                  (eq? (get table row 'name) 'ali)))
=> '(students (name id gpa) (ayse 2 3.7))
```

Here, we deleted rows whose `name` field is equal to `ali`.

### 3.8 (update-rows table predicate change)

Here `table` is a table, `predicate` and `change` are functions. This function calls `change` on each row that `predicate` returns true, and collects those values. It returns other rows as-is. An example:

```
> (update-rows student-table
                (lambda (table row)
                  (eq? (get table row 'name) 'ali))
                (lambda (table row)
                  (alter table row '((gpa 3.3)))))
=> '(students (name id gpa) (ali 1 3.3) (ayse 2 3.7)))
```

Here, we updated values of `gpa` field of rows whose `name` field equals to `'ali` to `3.3`.

### 3.9 (select-rows table fields predicate)

This function filters the rows by `predicate` then returns only fields given in `fields` list from table. Here `fields` is a list, and `predicate` is a predicate as described above. An example:

```
> (select-rows student-table
                '(gpa id)
                (lambda (table row)
                  (eq? (get table row 'name) 'ali)))
=> '(students (gpa id) (3.2 1))
```

Here, we created a table from GPA and ID number of students whose name is Ali.

### 3.10 (all-rows table row)

This function simply returns `true` for all rows, so it's mathematically equivalent to the predicate $P(t, r) : True.$. It is a predicate that is intended to be used for selecting all rows and some fields of a table. An example:

```
> (select-rows student-table
                '(gpa)
                all-rows)
=> '(students (gpa) (3.2) (3.7))
```

Here, we selected GPA of all students.

### 3.11 `(sort-by table field)`

Sorts the table by a given field in ascending order. You can use built-in `sort` function to define this function. For sorting symbols, you can use `symbol<?` function, for sorting strings you can use `string<?` function. You may need type-checking to create a generic comparison function. An example:

```
> (sort-by student-table 'id)
=> '(students (name id gpa) (ali 1 3.2) (ayse 2 3.7))
```

**An important clarification:** The cells in rows can be lists as well as symbols, numbers and strings. So, you need to order lists lexicographically, that is, a list $L$ is smaller than a list $M$ if and only if $\exists k \in \mathbb{N}, L_k < M_k \wedge (\forall i \in \mathbb{N}, i < k \implies L_i = M_i)$. Here, $L_i$ denotes $i$th element of $L$. Also you may assume that lists have same length to be comparable.

### 3.12 `(distinct table)`

Removes any duplicate rows in a table. Hint: you can sort the table first to find the duplicates easily and more efficiently (in $O(N \log N)$ rather than $O(N^2)$ time). An example:

```
> (distinct '(students (name id gpa) (ali 1 3.3) (ayse 2 3.4) (ali 2 3.3) (ali 1 3.3)))
=> '(students (name id gpa) (ali 1 3.3) (ayse 2 3.4) (ali 2 3.3))
```

## 4 Efficiency

Since you're going to write a kind of code that should be rather efficient in production environment, the code must be efficient, also there is a lot room for improvement (such as implementing tables with indexes implicitly), but they are not required for this project. You should still be sure that your methods run with a considerably large amount of data (say, about $N = 10^6$ rows for 10 tables) in a short amount of time (say, $t < 1s$), with simple comparison functions.

After all, nearly all of your functions, save `sort-by` and `distinct`, can have linear time complexity without using any complicated data structures such as trees, and some of them (such as `add-row`) should run in constant time in terms of number of rows.

## 5 Documentation and Clarity

Documenting the code is essential for developing in the long term and as you're not proficient in functional programming very much, you tend to solve the problems in rather obscure and/or unnatural ways. Thus, you should document every predicate in your project and you will be graded for documentation of your code in case your code becomes hard to understand.

You should add documentation to each function you defined (both the ones required by the project description ad the ones you've created as helpers) in the form below (which is sort of the de facto documentation format for Racket):

```
; (hypot x y) -> number?
; x : number?
; y : number?
;
; Calculates hypotenuse of the right triangle formed by x and y.
;
; Examples:
; > (hypot 3 4)
; => 5
; > (hypot 5 12)
; => 13
(define (hypot x y)
  (sqrt (+ (* x x) (* y y))))
```

Here first line is function's signature, second and third lines are parameter types, fifth line is function description and we have a few examples on lines 8-11.

Besides documentation, it is also important to write code that is readable, so avoid small, clever, hard-to-comprehend hacks (unless they fit the very nature of functional programming, in which case you should explain the hack in documentation) and unnecessary complexity in your code. You are also graded for code clarity. When you cannot find a clearer way to handle something, explain how your code works in documentation, as it is important for us to grade your project and to be sure that you understood the concepts behind functional programming.

# 6 Submission

You are going to submit a file explicitly named as `project-YOUR_STUDENT_ID.rkt` including the filename extension, all lowercase. The first 4 lines of the code must be in the form:

```
#lang racket
; compiling: yes
; complete: yes
; name surname
```

The first line declares language as Racket v6. The second line denotes whether your code is compiled correctly[1], the third line denotes whether you completed all of the project, which must be "no" (without quotes) if you're doing a partial submission. `name surname` must match with your name and surname and must be in *lowercase*. The four lines must be lowercase.

Check that you're submitting the code as described above. Explicitly check that,

1. The filename is correct, including the extension.
2. You submitted the right file, you can check that by downloading your submission from Moodle.
3. You didn't submit a zip file or something else.
4. The filename is all in *lowercase*.

[1] here to be compiled correctly means that you get no errors when you press the Run button to run your code

# 7 Prohibited Constructs

The following language constructs are ***explicitly prohibited***. You *will not get any points* if you use them:

1. Any function or language element that end with an `!`.
2. Any of these constructs: `begin`, `begin0`, `when`, `unless`, `for`, `for*`, `do`, `set!-values`.
3. Any language construct that starts with `for/` or `for*/`.
4. Any construct that causes any form of mutation[2] (impurity).

# 8 Some Tips on The Project

- You can check out PS slides.
- You can (and should) use higher-order functions like `map`, `filter`, `foldl` and `foldr`. You are also encouraged to use anonymous functions with help of `lambda`. The functions in project are designed to make use of them.
- You may need to define your own improvements (such as defining an extended `cons`) to write some functions more easily.
- The variadic functions will be explained in next PS. A nice explanation is on http://stackoverflow.com/a/12942215/1162734.
- Contracts make your code easier to debug, they are promises made between caller and callee in a function call, and they reduce time for testing and debugging. They are used in many dynamic programming languages, to improve checking at runtime. You can learn more about them at http://docs.racket-lang.org/reference/contracts.html.
- You can use Racket reference, either from DrRacket's menu: Help > Racket Documentation, or from the following link: http://docs.racket-lang.org/reference/index.html
- SICP (Structure and Interpretation of Computer Programs a.k.a. The Purple Book) is a nice book for learning Scheme: https://mitpress.mit.edu/sicp/full-text/book/book.html
- There are also video lectures using purple book on MIT OCW, by the authors: http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-001-structure-and-interpretation-of-computer-programs-spring-20 video-lectures/

[2] Mutation means creating any kind of change, either via changing value of a variable, a memory slot, or an element of a container; or via doing input/output.