

# Project 1

**Due:** Both the paper and the electronic submission are due on **Thursday, June 21, 6 pm, for the day section**, and **Friday, June 22, 6 pm for the evening section**, in 324 drop box (SF, 2nd floor, near bridge to LP).

**Lateness reminder:** penalty of 15% of the mark if one day late, 30% if two days late; not accepted later.

**Note:** The names of your functions should be the ones required in this assignment. Otherwise you will loose marks. The name of the submitted file should be **p1.scm**. Otherwise you will loose marks.

In this project you will write pure functional code, i.e., you should not use loop, do, set! and other functions ended in !. In short you should not use extensions borrowed from the imperative style languages. You can use let or let\* if you think it is really necessary (but you can always use helper function to avoid it).

## 1 Part A

Define functions for operations with multisets. A multiset is a set of elements which can contain multiple occurrences of an element. The order of occurrence is not important. An example is: { a b c a c c }.

We represent a multiset as a list. Each element of the list will be a dotted pair (*< element > . < counter >*). The counter is the number of the occurrences of that element (> 0). For example the previous multiset with two a's, one b, and three c's is:

```
((a . 2) (b . 1) (c . 3))
```

1. Define a function **m\_union** which computes the union of two multisets. If an element is present in both multisets, the union will contain the element with the counter equal to the sum of the counters in the two multisets.

```
1 ]=> (m_union '((a . 2) (b . 1) (c . 3)) '((b . 3) (d . 1)))
;Value 1: ((a . 2) (b . 4) (c . 3) (d . 1))
```

2. Define a function **m\_intersection** which computes the intersection of two multisets. If an element is present in both multisets, the intersection will contain the element with the counter equal to the minimum of the counters in the two multisets. If an element is not in both multisets, it won't be in the intersection.

```
1 ]=> (m_intersection '((a . 2) (b . 1) (c . 3)) '((a . 1) (b . 3) (d . 1)))
;Value 2: ((a . 1) (b . 1))
```

3. Define a function **list2mset** which transforms a list in a multiset. Example:

```
1 ]=> (list2mset '(a b c a c c))
;Value 3: ((a . 2) (b . 1) (c . 3))
```

## 2 Part B

In this part of the project you will extract information from a database of students and a database of courses.

We encode the database of students as a list of entries, with one entry for each student. Each entry is a list having as first element the name of the student, and as a second element a list of courses the student took. A student name is simply a symbol with the first name of the student. A course name is simply a symbol such as 'csc324.

```
( < student > ( < course1 > < course2 > ... < coursen > ) )
```

We encode the database of courses as a list of entries, with one entry for each course. Each entry is a list having as first element the name of the course, and as a second element a list of prerequisites.

```
( < course > ( < prereq1 > < prereq2 > ... < prereqn > ) )
```

A prerequisite can be a course (a symbol) or a list of alternative prerequisites. For example, the fact that csc324 has the prerequisites csc148 and one of csc238 and mat246 is expressed as: (csc324 (csc148 (csc238 mat246))). A list of alternative prerequisite contains two or more alternatives.

We use the *define* statement to associate a name with the representation of the database. The Scheme code for a *sample* database is below. Your program should be able to run on any database using the same representation. **Do not** define any other global variables except STUDENTS and COURSES in your program. The code below is given as starter code:

```
(define STUDENTS
  '((jim ())
    (jane (mat101 csc108) )
    (john (mat101 csc108 csc148 csc270) )
    (emily (mat101 csc108 mat246) )
    (emma (mat101 csc108 csc148 mat246) )
  )
)

(define COURSES
  '((csc108 ())
    (csc148 (csc108 mat101) )
    (csc238 (csc148) )
    (csc260 (csc108) )
    (csc270 (csc148) )
    (csc310 ((csc260 csc270) (sta107 sta205 sta255)) )
    (csc324 (csc148 (csc238 mat246)) )
  )
)
```

1. Write the following functions:

(a) **taken**: has a single argument (a student name) and returns all courses taken by that student. Example:  
1 ]=> (taken 'emma)  
;Value 1: (mat101 csc108 csc148 mat246)

(b) **prereq**: has a single argument (a course name) and returns the prerequisites of that course course, in the format they are given in the database. Example:

```
1 ]=> (prereq 'csc324)
;Value 2: (csc148 (csc238 mat246))
```

HINT: These functions take a single argument, but they need to step through the STUDENTS or COURSES database using recursion. This means that you will need a helper function to recursively step through the databases.

2. Define a function named **can-take**, which takes two arguments, a student name and a course, and returns true if the student has the prerequisites to take the course (returns false otherwise). If there are alternative prerequisites, the student is required to have one of them (at least one). A student cannot take a course if she/he already took it. Example:

```
1 ]=> (can-take 'emma 'csc324)
;Value: #t
```

3. The following functions will be graded on your use of higher order functions to accomplish the task. Some of the functions can easily be written using calls to function you have written above. You may need additional helper functions to “clean-up” the results. The results should **not contain duplicates**, and the order of the elements in the result does not matter. For this question you don’t have to worry so much about efficiency, but about short and elegant code.

(a) **all-students**: takes no arguments and returns a list with the names of all students enumerated in the database STUDENTS. Example:

```
1 ]=> (all-students)
;Value 3: (jim jane john emily emma)
```

(b) **all-courses-taken**: takes no arguments and returns a list with all the courses taken by at least one student. Example:

```
1 ]=> (all-courses-taken)
;Value 4: (csc270 mat101 csc108 csc148 mat246)
```

(c) **can-take-list**: takes two arguments, a student and a list of courses, and returns true if the student can take all the courses in the list (false otherwise). Example:

```
1 ]=> (can-take-list 'emma '(csc324 csc270))
;Value: #t
```

(d) **all-courses-and-prereq** takes no arguments and returns a list with all the courses in the database COURSES, including the ones mentioned in prerequisites part. Example:

```
1 ]=> (all-courses-and-prereq)
;Value 5: (mat101 csc108 csc310 csc260 csc270 sta107 sta205 sta255 csc324 csc148 csc238 mat246)
```

### 3 What to hand in?

#### 3.1 On paper

Please use an unsealed envelope, having on top the cover page provided at the end of this file. Put inside:

- A printout of your code.
- A printout of one or more script files containing tests run for each function. (See `man script` if you are unsure of how to create a script file.) The tests for each function you implemented should include several relevant runs, degenerated cases for the input (such as empty list). You can save your test cases in a file, then cut and paste when you use `script`. You are allowed to try alternative methods Scheme may provide for producing script files. For example, a simple method is to use:  
`scheme <file1 >test_result_file`, where `file1` contains `(load "p1")` and your test cases.

### 3.2 Electronically

In addition to your paper submission, you must submit your code electronically.

- Put all definitions of the functions above, as well as any helper function, in a file called `p1.scm`
- Submit the file `p1.scm` using the following command:  
`submit -N project1 csc324h p1.scm`

See `man submit` if you need more information. Note that if you already submitted a file and you wish to replace it with a different version, you can submit it again. But you must use the `-f` option to overwrite the old version.

## 4 How the project will be marked?

It is worth emphasizing that your project will be **marked not only for correctness, but for functional programming style, comments, and your testing strategy as well.**

**Style:** the code should use functional programming not imperative style. Helpers functions should be used when appropriate. Significant names should be used for functions and their arguments. The code should be as efficient and elegant as possible.

**Comments:** should clearly state what are the preconditions of each function (expected number of arguments and their type), and what are the postconditions (results and expected behaviour). Additional comments inside the functions should be used to explain steps that are not obvious. The comments should consist of full English sentences, including punctuation.

**Testing:** all functions should be tested (including helpers). The test cases for each function should include relevant cases (no less and no more than the necessary cases), including degenerate cases such as empty lists. Please explain in a few sentences what is your testing strategy for each question.

## Cover Page for Project 1

Family name: \_\_\_\_\_ Student #: \_\_\_\_\_

First name: \_\_\_\_\_ CDF id: \_\_\_\_\_

Section:        Day section        Evening section

I declare that this assignment is my own work, and it is in accordance with the University of Toronto Code of Behaviour on Academic Matters and the Code of Student Conduct. I declare that I didn't discussed this assignment with anybody else.

Signature \_\_\_\_\_

The following is for our use in grading:

Q1 \_\_\_\_\_ / 24

Q2 \_\_\_\_\_ / 40

Style \_\_\_\_\_ / 8

Comments \_\_\_\_\_ / 8

Testing \_\_\_\_\_ / 20

\_\_\_\_\_

Total \_\_\_\_\_ / 100

Other (penalties)

Final mark \_\_\_\_\_ / 100