

<b>COMP1811</b> <b>(2022/23)</b>	<b>Paradigms of Programming</b>	<b>Header ID</b>	<b>Contribution:</b> 20% of module
<b>Module Leader/Moderator</b> Yasmine Arafa/Andy Wicks	Practical Coursework – Scheme Project		<b>Deadline Date</b> Monday 20/03/2023
This coursework should take an average student who is up to date with tutorial/lab work approximately <b>7 hours</b> .  Feedback and grades are normally made available within 15 working days of the coursework deadline.			
<b>Learning Outcomes:</b>  A. Understand the programming paradigms introduced and their applicability to practical problems. B. Apply appropriate programming constructs in each programming paradigm. C. Design, implement and test small-scale applications in each programming paradigm. D. Use appropriate tools to design, edit and debug programs for each paradigm.			

**Plagiarism** is presenting somebody else's work as your own. It includes copying information directly from the Web or books without referencing the material; submitting joint coursework as an individual effort; copying another student's coursework; stealing coursework from another student and submitting it as your own work. Suspected plagiarism will be investigated and if found to have occurred will be dealt with according to the procedures set down by the University. Please see your student handbook for further details of what is/is not plagiarism.

**All material copied or amended from any source (e.g. internet, books) must be referenced correctly according to the reference style you are using. Code snippets from open-source resources or YouTube must be acknowledged appropriately.**

**Your work will be submitted for electronic plagiarism checking. Any attempt to bypass our plagiarism detection systems will be treated as a severe Assessment Offence.**

## Coursework Submission Requirements

- An **electronic copy** of your work for this coursework must be fully **uploaded by 23:30 on Monday 20/03/23**. Submissions must be made using the Scheme Project Upload link under "CW Details and Submission" on the Moodle page for COMP1811. **Your work will be capped if you fail to submit by the deadline.**
- For this coursework you must submit 3 files: a zip file containing the code files required to run **your Scheme project**, a PDF version of **your report**, and a screencast showing your code running. In general, any text in the report **MUST NOT** be an image (i.e. must not be scanned or a screenshot) and would normally be generated from other documents (e.g. MS Office using "Save As PDF"). **Marks will be deducted if you include images** instead of code or text. **There are limits on the file size.**
- Make sure that any files you upload are virus-free and not protected by a password or corrupted, otherwise they will be treated as null submissions.
- Your work will be marked online, and feedback and a provisional grade will be available on Moodle. A news item will be posted when the feedback is available and when the grade is available in BannerWeb.
- All coursework must be submitted as above. Under no circumstances can they be accepted by academic staff.

The University website has details of the current Coursework Regulations, including details of penalties for late submission, procedures for Extenuating Circumstances, and penalties for Assessment Offences. See <http://www2.gre.ac.uk/current-students/regs>

## Coursework Regulations

- If you have Extenuating Circumstances, you may submit your coursework up to 10 working days after the published deadline without penalty, but this is subject to your claim being accepted by the Faculty Extenuating Circumstances Panel.
- Late submissions will be dealt with in accordance with University Regulations.
- Coursework submitted more than two weeks late may be given feedback but will be recorded as a non-submission regardless of any extenuating circumstances.
- Do not ask lecturers for extensions to published deadlines - they are not authorised to award an extension.

Please refer to the University Portal for further detail regarding the University Academic Regulations concerning Extenuating Circumstances claims.

## Coursework Specification

Your overall task for the COMP1811 Scheme Project Coursework is to design and develop a small program in Scheme. You are expected to work individually to develop this project.

Implementing the program will give you experience in the development of an application using Functional Programming, where you are expected to design, implement, and use your own code from scratch. Your code development is the culmination of all that you have learned and all your hard work for part two of this module (Functional Programming).

**Your solutions are expected to be integrated into the code template provided. Please, stick to the template given to you and complete the Scheme definitions with your own code.**

**It is recommended to switch off your mobile. Programming requires mindfulness.**

**Please read the entire coursework specification.**

## Scenario

You are programming your first App in Scheme: “*Restoring Facebook Social Network after it was hacked!*”.



Figure 1 Facebook network as a graph

## Description of Coursework

Mark Zuckerberg has two reasons to worry. **Facebook**© datacentre has been hacked and apparently, the software has been partly corrupted.

Consultants have identified the damaged parts of the software and produces the **Table 1** Damage report – see Table 1 below. Luckily, the software was written in *Scheme*, using the *Functional Programming* paradigm. So, the software can be very easily repaired in isolation and restored despite the overall complexity of the system. That is one of the main principles of FP, *referential transparency*, or in Leibniz’s words: “*Equals replace equals*”.

You are expected to repair individual definitions in the given code template. Let the force be with you, and *don’t forget to format your code with Ctrl+i* !

## Task 0

Familiarise yourself with the given code.

- 1) Acquire the source code from [sn-scheme-template.zip](#) on Moodle.
- 2) Download, unzip, and run it in **DrRacket** (open **sn-app.rkt** and run).
- 3) **testing.rkt** is a test harness that runs the definitions that you will need to complete. As you implement more and more functions, the template will work as expected. See the [demo video on Moodle](#) showing the individual commands that the make use of the definitions you will complete.
- 4) Locate the files and the functions you are expected to implement (check **Damage report** – Table 1 below).
- 5) Please DO NOT alter the rest of the definitions in sn-app.rkt. You may read but not modify them.

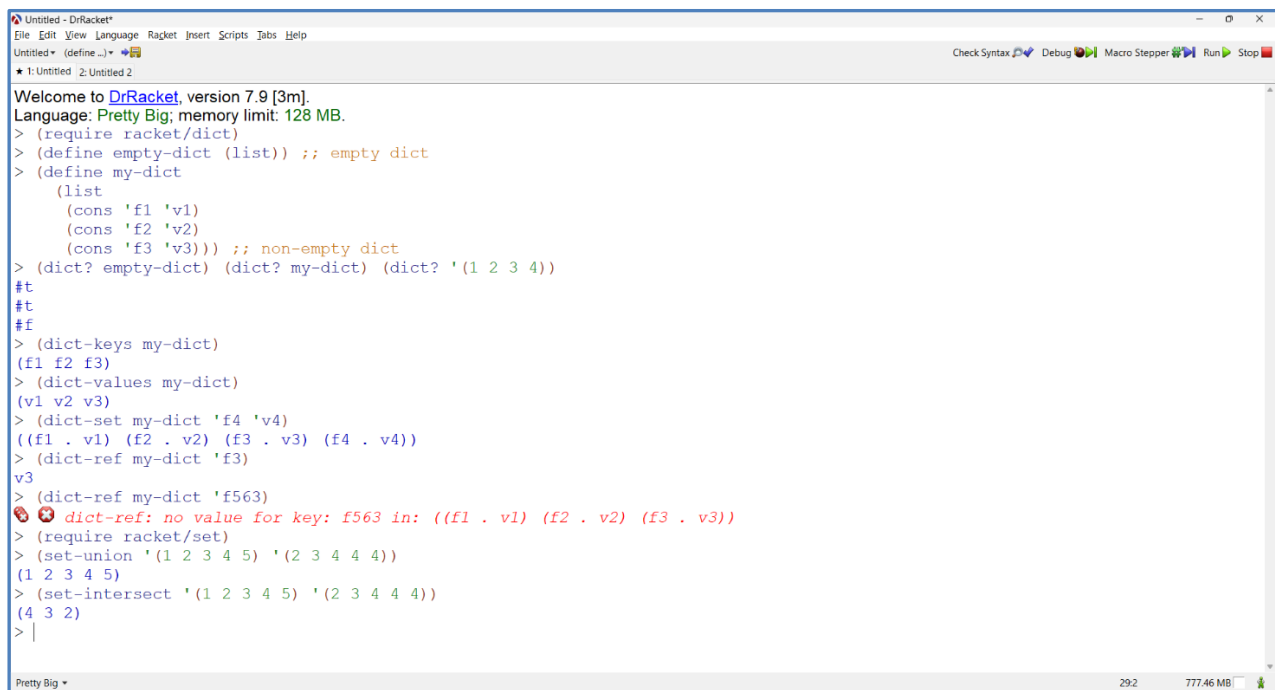
You must program using only *pure, side-effects free functions, using recursion or high-order programming*. In practical terms, you cannot use **set!**, **printf**, **read**, **for-each**, **begin** or any other side effect functions in Scheme<sup>1</sup>. Function with side effects conventionally have an exclamation mark (!) as the final character of the function name.

### Fundamentals.

A **social network** - see Figure 1, can be viewed in mathematical terms as a **graph** (network), made of **nodes** (users) and **edges** (links) among them. Essentially you can:

- add users (**nodes**)
- connect friends (link **nodes** by **edges**)
- count the number of users (count how many **nodes**)
- count the number of friends a user has (count how many **edges** a **node** has)
- computing the friendliest (unfriendliest) user (**node** having max(min) number of **edges**).
- others...

It may be implemented using **lists** or **dictionaries**. In **Scheme**, everything is a list, so a **dictionary** is nothing but a **list of pairs entry-value**.



```
Welcome to DrRacket, version 7.9 [3m].
Language: Pretty Big; memory limit: 128 MB.
> (require racket/dict)
> (define empty-dict (list)) ;; empty dict
> (define my-dict
  (list
    (cons 'f1 'v1)
    (cons 'f2 'v2)
    (cons 'f3 'v3))) ;; non-empty dict
> (dict? empty-dict) (dict? my-dict) (dict? '(1 2 3 4))
#t
#t
#f
> (dict-keys my-dict)
(f1 f2 f3)
> (dict-values my-dict)
(v1 v2 v3)
> (dict-set my-dict 'f4 'v4)
((f1 . v1) (f2 . v2) (f3 . v3) (f4 . v4))
> (dict-ref my-dict 'f3)
v3
> (dict-ref my-dict 'f563)
dict-ref: no value for key: f563 in: ((f1 . v1) (f2 . v2) (f3 . v3))
> (require racket/set)
> (set-union '(1 2 3 4 5) '(2 3 4 4 4))
(1 2 3 4 5)
> (set-intersect '(1 2 3 4 5) '(2 3 4 4 4))
(4 3 2)
> |
```

**RATIONALE:** A **node** can be viewed as the **key**, and the list of friends as the **value** for that key<sup>2</sup>.

<sup>1</sup> You may use it for debug purposes, dropping them in final release.

<sup>2</sup> Note dictionaries are useful in finding the *intersection and unions between lists*.

Table 1 Damage report

MODULE	FUNCTION
<b>sn-app.rkt (ROM)</b>	Undamaged – no changes required. <b>Do not alter.</b>
<b>sn-console.rkt (ROM)</b>	Undamaged – no changes required. <b>Do not alter.</b>
<b>sn-io.rkt (ROM)</b>	Undamaged – no changes required. Undamaged – no changes required. Do not alter.
<b>sn-graph.rkt</b>	(Contains damaged functions)
<b>sn-social-network.rkt</b>	(Contains damaged functions)
<b>sn-utils.rkt</b>	(Contains damaged functions)

## Task 1

The following are the specifications of the functions that need to be *repaired*. Be aware that functions can use existing functions, i.e., they can call other functions that have been defined or found from libraries.

**The following are the damaged files for you to correct:-**

### File 1 - sn-utils.rkt (Damaged)

#### i. sn-list->dict (1 mark)

Given a list of entries, this function should return the argument passed. This function creates a list of pairs.

```
> (sn-list->dict (list (cons 'k1 'v1) (cons 'k2 'v2)))  
((k1 . v1) (k2 . v2))
```

#### ii. sn-dict-ks-vs (2 marks)

Given a **list** of **values** and a list of user IDs/Names (**keys**), this function should return a **dictionary** (remember - a list of keys with their corresponding values).

```
> (sn-dict-ks-vs '(k1 k2 k3) '(v1 v2 v3))  
((k1 . v1) (k2 . v2) (k3 . v3))
```

#### iii. sn-line->entry (4 marks)

Given a **string**, this function should first split it into a list, then return a **dictionary entry**, i.e a **list** where the first element is the **key** and the remainder of the list is the corresponding **value** as **symbols**<sup>3</sup>.

```
> (sn-line->entry "A B C D")  
(A B C D)
```

---

<sup>3</sup> The following procedures: **string-split** and **string->symbol** may be used here. You will need to add (**require racket/string**) to your code to import those procedures.

## File 2 - sn-graph.rkt (Damaged)

The template given to you on Moodle, **sn-scheme-template.zi**, contains **testing.rkt** which can be used to test all your functions.

```
> (define my-dict
  (list
    (cons 'f2
      (list 'f3 'f4))
    (cons 'f3
      (list 'f2))
    (cons 'f4
      (list 'f3 'f2))
    (cons 'f13
      (list ))
    (cons 'f1
      (list ))
  ))
```

```
> my-dict
((f2 f3 f4) (f3 f2) (f4 f3 f2) (f13) (f1))
```

### i. sn-empty (1 mark)

Define the constant named *sn-empty* as the empty list. *sn-empty* defines the **empty** network.

```
> sn-empty
()
```

### ii. sn-users (2 marks)

This function returns the **list of users** (IDs or names) in the given network, *my-dict*.

```
> (sn-users my-dict)
(f2 f3 f4 f13 f1)
```

### iii. sn-add-user (5 marks)

Given a network and an ID or name, this function should add a new entry to the network for that user. Before you add the user you should check whether that user already exists and if they do return the network unchanged.

```
> (sn-add-user my-dict 'f5)
((f2 f3 f4) (f3 f2) (f4 f3 f2) (f13) (f1) (f5))
```

### iv. sn-add-frndshp (5 marks)

Given a network and two existing **keys** (user IDs or names), this function should update both users' list of friends.

```
> (sn-add-frndshp my-dict 'f1 'f2)
((f2 f1 f3 f4) (f3 f2) (f4 f3 f2) (f13) (f1 f2))
```

### File 3 - sn-social-network.rkt (Damaged)

#### i. sn-ff-for (2 marks)

Given a **network** and an existing ID/name, this function should return the list of friends for that user ID/name.

```
> (sn-ff-for my-dict 'f4)
(f3 f2)
```

#### ii. sn-cmn-frnds-btwn (6 marks)

Given a network and two existing IDs/names, this function returns the list of friends of friends in common between these two users.

```
> (sn-cmn-frnds-btwn my-dict 'f2 'f4)
(f3)
```

#### iii. sn-frnd-cnt (4 marks)

Given a network, this function returns a list of pairs. Each pair contains a user ID/name and the number of their friends.

```
> (sn-frnd-cnt my-dict)
((f2 . 2) (f3 . 1) (f4 . 2) (f13 . 0) (f1 . 0))
```

#### iv. sn-frndlst-user (4 marks)

Given a non-empty network, this function returns a pair consisting of the user ID/name of the person having the maximum number of friends in the network and the number of friends that they have.

```
> (sn-frndlst-user my-dict)
(f2 . 2)
```

#### v. sn-unfrndlst-user (4 marks)

The same as previous, but with minimum number of friends.

```
> (sn-unfrndlst-user my-dict)
(f1 . 0)
```



## Task 2

Write a **brief report** describing the main FP topics (one statement in plain English, please) you used to implement each function (i.e., *recursion*, *high-order*, *lists*, *union*, *intersection*, *dictionaries*, *meta-programming*...).

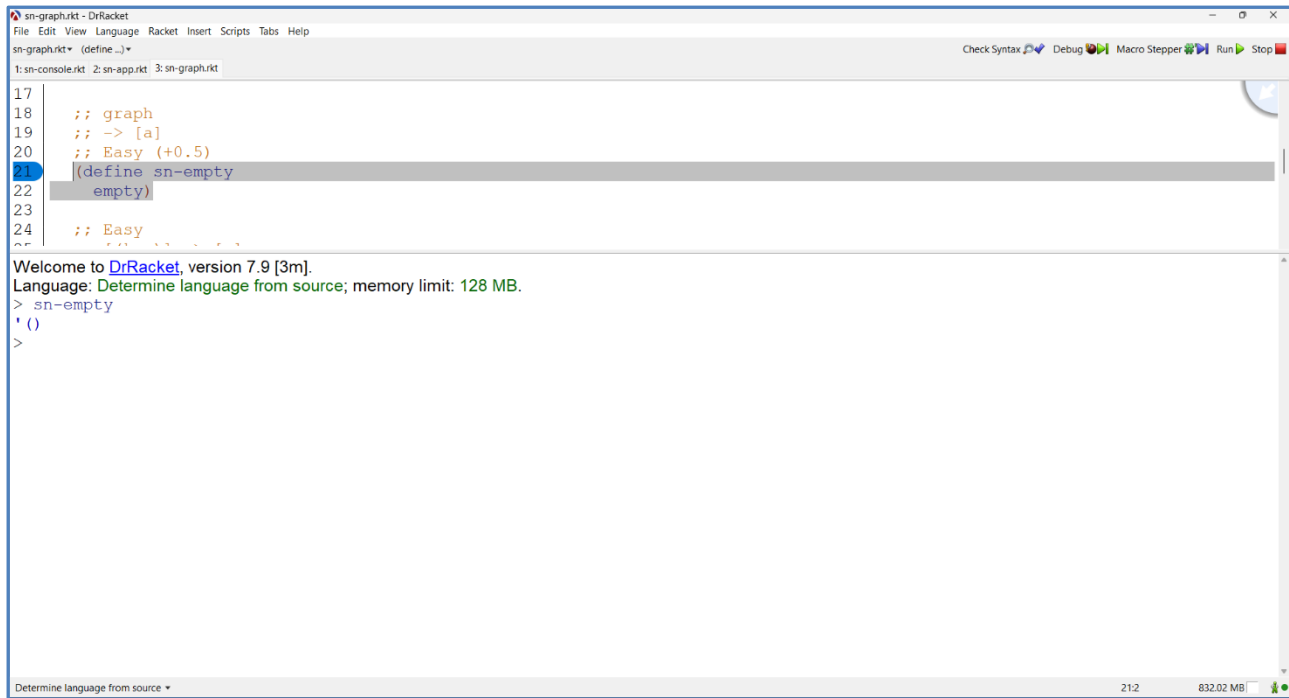
Upload the folder containing the **Scheme code you developed (.rkt files)**, **your final report as a pdf file**, **and a 5–10-minute screencast (explaining the code and showing your system running)** to Moodle under the Scheme Project Upload link under "CW Details and Submission" **by 23:30 on 20/03/2023**.

You are also required to **demo your work** during your scheduled lab session in the week commencing 20/03/23 in agreement with your lab tutor. You may fail this assessment if you fail to attend the demonstration without extenuating circumstances.

## How to start coding

Start by fixing the **sn-empty** function (item 1) in Damage report.

1. Open “sn-utils.rkt” file in one Racket tab.
2. Once coded, re-run the module, and from REPL, just type what you want to check.



Remember: To run the console, open “**sn-app.rkt**” and make it run. It is not mandatory to make it run to get a pass.

## Marking Scheme (mark breakdown)

Functionality			Demo and Video	10
F1 i	1		Testing	5
F1 ii	2		Evaluation	5
F1 iii	4		Quality of code	25
F2 i	1		Documentation	15
F2 ii	2			
F2 iii	5		<b>OVERALL</b>	100
F2 iv	5			
F3 i	2			
F3 ii	6		The demo and video are mandatory.	
F3 iii	4			
F3 iv	4			
F3 v	4			
	40			

## Grading Criteria

Grading Criteria Judgement for the items in the marking scheme will be made based on the following criteria. To be eligible for the mark in the left-hand column you should at least achieve what is listed in the right-hand column. Note that you may be awarded a lower mark if you do not achieve all the criteria listed. For example, if you achieve all the criteria listed for a 2:1 mark but have a poor report then your mark might be in the 2:2 range or lower.

> 80%

*Exceptional*

- Completed functionality implemented and tested to an outstanding standard or above (all required features implemented, no obvious bugs, outstanding code quality, Scheme language features accurately applied).
- Able to show outstanding, thorough knowledge and systematic understanding of functional programming concepts and Scheme language features in the code and the report.
- Overall report – outstanding (required sections completed accurately and clearly, easy to read, structured and coherent and insightful arguments for design justification and use of Scheme language).
- Evaluation section of report – outstanding (insightful points made relevant to development (system produced), productivity, errors, learning, and time management; thorough introspection, realistic and insightful reflection).

70-79%

*Excellent*

- Completed functionality implemented and tested to an excellent standard (required features implemented, no obvious bugs, excellent code quality, Scheme language features accurately applied).

	<ul style="list-style-type: none"> <li>• Able to show excellent, detailed knowledge and systematic understanding of functional programming concepts and Scheme language features in the code and in the report.</li> <li>• Overall report – excellent (required sections completed accurately and clearly, easy to read, well justified design decisions and clear argument, comparative reasoning).</li> <li>• Evaluation section of report – excellent (interesting points made relevant to development (system produced), productivity, errors, learning, and time management; detailed introspection and interesting reflections).</li> </ul>
60-69% <i>Very Good</i>	<ul style="list-style-type: none"> <li>• Completed functionality implemented and tested to a very good standard (required features implemented, few if any bugs, very good code quality, very good attempt using Scheme language features, may contain some design flaws).</li> <li>• Able to show very good knowledge and systematic understanding of functional programming concepts and Scheme language features in the code and in the report.</li> <li>• Overall report – very good (required sections completed accurately and clearly, good argument for design justification and use of Scheme).</li> <li>• Evaluation section of report – very good (very good points made relevant to at least three out of the following: development (system produced), productivity, errors, learning, and time management; introspection shows some reasonable thought).</li> </ul>
50-59% <i>Good</i>	<ul style="list-style-type: none"> <li>• Completed functionality implemented and tested to a good standard (required features implemented, few if any bugs, good attempt using Scheme language features, contains some design flaws).</li> <li>• Able to show good knowledge and mostly accurate systematic understanding of functional programming concepts and Scheme language features in the code and in the report.</li> <li>• Overall report – good (required sections completed accurately, mostly clear, some reasonably balanced argument for design justification and use of Scheme).</li> <li>• Evaluation section of report – good (addresses points relevant to at least three out of the following: development (system produced), productivity, errors, learning, and time management; limited introspection).</li> </ul>
45-49% <i>Satisfactory</i>	<ul style="list-style-type: none"> <li>• Completed functionality implemented and tested to a good standard (some features implemented, few if any bugs, acceptable attempt using Scheme language features, contains some design flaws).</li> <li>• Able to show satisfactory knowledge of functional programming concepts and Scheme language features in the code and in the report.</li> <li>• Overall report – acceptable (required sections completed, mostly accurate and clear, superficial justification for design choices and use of Scheme).</li> <li>• Evaluation section of report – acceptable (addresses points relevant to at least two out of the following: development (system produced), productivity, errors, learning, time management, mostly descriptive; superficial introspection).</li> </ul>
40-45% <i>Satisfactory</i>	<ul style="list-style-type: none"> <li>• Completed functionality implemented and tested to a reasonable standard (at least 2 required features implemented with a few minor bugs, acceptable attempt using Scheme language features, contains some design flaws).</li> <li>• Able to show fairly satisfactory knowledge of functional programming concepts and Scheme language features in the code and in the report.</li> <li>• Overall report – acceptable (required sections completed, mostly accurate, clumsy language, descriptive account of design choices and use of Scheme).</li> </ul>

	<ul style="list-style-type: none"> <li>• Evaluation section of report – acceptable (addresses points relevant to at least one out of the following matters: development (system produced), productivity, errors, learning, time management, mostly descriptive; superficial introspection).</li> </ul>
35-39% <i>Fail</i>	<ul style="list-style-type: none"> <li>• Good attempt at some features although maybe buggy with some testing.</li> <li>• Confused knowledge of functional programming concepts and Scheme language features in the code and in the report.</li> <li>• Overall report – mostly completed to an acceptable standard. Some sections are confused.</li> <li>• Evaluation section of report – mostly descriptive but showing some thought.</li> </ul>
<35% <i>Fail</i>	<ul style="list-style-type: none"> <li>• Little or no attempt at features or very buggy with little or no testing.</li> <li>• Not able to show satisfactory knowledge functional programming concepts and Scheme language features in the code and in the report.</li> <li>• Overall report – mostly in-completed, at an un-acceptable standard or missing.</li> <li>• Evaluation section of report – descriptive showing a lack of thought or missing.</li> </ul>