Practical Session 3: **Scheme** (List processing)

1 Getting started

There have been many LISP dialects throughout history. For the purpose of this course, we will use one of the best-known of these dialects: Scheme. Scheme itself is language which is described in a standard called *Revised^n Report on the Algorithmic Language Scheme (RnRS)*[1], with n the version number of the document. Although the document is still revised periodically to this day (with R7RS published in 2013), the most commonly implemented version of this standard is R5RS.

One general-purpose implementation is available in Racket[2]. Racket is actually the name of a LISP dialect itself, but the racket programs allow to use both the Racket and Scheme R5RS dialects, as well as some other variants. The package provides a command-line interpreter as well as a GUI called DrRacket. In the GUI, you can write a script in the top panel. The bottom panel is an interpreter in which you can write expressions directly, or see the result of running the script typed in the top panel.

Every source file in DrRacket should start with an expression of the form `#lang language` where language tells the interpreter which dialect you want to use. In our case, we will mostly use `#lang r5rs`. If you wish to use your own text editor, you can run your programs with `racket -r myscript.scm`.

2 The basics

Scheme uses prefix notation. This means that applying a function f to two arguments x and y is done with the following:

```
(f x y)
```

For instance, to compute $2 * (14 + 17)$, you will write:

```
(* 2 (+ 14 17))
```

Write and evaluate the following expression in Scheme:

$$\frac{(134 + 121) * \frac{74}{\sqrt{8}}}{97.1 - \frac{exp(7)}{78*71}}$$

---

[1] http://www.schemers.org/Documents/Standards/R5RS/
[2] https://racket-lang.org/

Even flow control uses such a notation. For instance, an `if` condition would be written as:

```scheme
(if test-expr true-expr false-expr)

(if (positive? a) "a is positive" "a is negative")
```

Try to re-write the following Pascal piece of code in Scheme:

```pascal
        a := -8;

        if a > 0 then b := 2 * sqrt(a)
        else b := 7 * sqrt(-a) / -a;

        writeln(b)
```

You can start from the following skeleton to get you started:

```scheme
        (define a -8)
        (define b ... )
        (writeln b)
```

---

3 Lists

Scheme is a dialect of LISP, which stands for *List Processor* and lists are at the core of LISP languages. They are the main data structure and the language is designed to manipulate them in many ways, and use them to define more complex data structures.

The basic building blocks of Scheme are *atoms*. They can be numbers, symbols (`'a`, `'mysymbol`), characters (`#\a`, `#\Z`), booleans (`#t` and `#f` for true and false),...

Atoms can be combined together with the `cons` function to form a *pair*. This function takes two argument and returns a pair, in the which the first item is the first argument and the second item is the second argument. The Scheme interpreter will write a pair as:

```scheme
        (a . b)
```

Given a pair, obtaining the first and second elements of the pair can be done with the infamous `car` and `cdr` functions.

```scheme
(define mypair (cons 'a  'b))
(car mypair)        ; -> Returns the symbol 'a
(cdr mypair)        ; -> Returns the symbol 'b
```

A *list* is a pair, in which the second element (the `cdr`) is a list or the constant `null` (also written `'()`). For instance, this defines a list :

```
(define mylist (cons 'a  null))
(list? mylist)          ; -> returns #t, true
```

You can see that a list has a recursive definition : a list is a pair where the second element is a list. The empty list '() gives us a base case from which to build lists. Scheme provides facilities to build lists straightaway more easily than using multiple cons. For example, both of these lines define valid lists:

```
'(22 21 23 #f 45 0 'a  "lists can have heterogeneous elements")

(list 1 2 3 4 "A cat is fine too")
```

Attention: because Scheme uses mutable lists and racket uses immutable lists (we will talk about list mutability in a following session). For this reason, when using the R5RS standard, racket will sometimes show lists like this

```
(list 1 2 3)
-> (mcons 1 (mcons 2 (mcons 3 '())))
```

This is because racket uses mcons instead of cons for mutable lists. To avoid this problem, always use the display function to print out lists.

Try to guess the value of the following expressions. Then verify your answer in racket.

```
        (cons 1 (cons 2 (cons 3 (cons 4 '() ))))
        (car (cons 1 (cons 2 '())))
        (car (cdr (list 1 2 3 4)))

        (cadr '(1 2 3))
        (caddr '(1 2 3))
        (cdddr '(1 2 3))

        (cons (cons 'a 'b) (cons 1 '() ))
        (list (list 1 2 3) 4 5 6)

        (caddr '((1 4 8 7) (1 3 4 2 5) (0 32 (54 4 46))))
        (caaddr '((1 4 8 7) (1 3 4 2 5) (0 32 (54 4 46))))
        (cadar '((1 4 8 7) (1 3 4 2 5) (0 32 (54 4 46))))
```

What is the difference between these two objects?

```
        (cons 'a (cons 'b 'c))
        (cons 'a (cons 'b (cons 'c '() )))
```

---

4 Functions

Scheme follows the functional programming paradigm. In the next session, we will see what this means and how functions in Scheme can be passed around and manipulated as first order items.

For now, we will just explore the basic syntax to define functions in Scheme and how many problems in Scheme are defined recursively. Look at the following example then try to implement some more complex functions listed below.

```
(define (cube x) (* x x x))
```

4.1 Recursion on numbers

Write a function `sum-cubes` of two integer arguments a and b ($a < b$) which computes the sum of cubes of the integers in $\{a, a + 1, ..., b\}$. For instance (`sum-cubes` 0 3) should return 36 ($0^3 + 1^3 + 2^3 + 3^3 = 36$).
Hint: $sumCubes(a, b) = a^3 + sumCubes(a + 1, b)$

```
(define (sum-cubes a b)
        ...
        )
```

Write a recursive function `collatz?` which checks the Collatz conjecture for an integer n. Print all the steps of the check along the way (use `writeln` for instance). Remember from the previous weeks that the Collatz conjecture states that the following process converges to 1:

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n + 1 & \text{if } n \text{ is odd} \end{cases}$$

Hints: You may find the `begin` and `cond` builtins useful. Scheme also has the `even?` and `odd?` predicates built-in. As a side-exercise, can you define versions of these predicates yourself?

4.2 Tail recursion

Scheme implements tail-recursion. To put it in simple words, tail-recursion occurs whenever a recursive function calls is located at the end of the function: in this situation, the control does not need to go back to the caller function to the callee, because there is nothing left to execute in the caller function. Because of this, the recursive calls can actually be implemented as a loop under the hood, avoiding the performance issues that recursive functions can suffer (filling up the call stack). See the racket doc[3] and the R5RS section 3.5 for more details about tail recursion.

---

[3] https://docs.racket-lang.org/guide/Lists__Iteration__and_Recursion.html#%28part._tail-recursion%29

Is the `sum-cubes` function you implemented above tail-recursive? Why? If not, can you make it tail-recursive?
Compare the performance difference between both implementations with:

```
(time (sum-cubes 0 999999))
(time (sum-cubes-tail 0 999999))
```

The `time` function is not a Scheme primitive, but a Racket extension, so you will need to change your dialect to `#lang racket` for this to run. Notice how racket handles big numbers naturally.

4.3 Recursion on lists

Using the `car` and `cdr` primitives, you can access the head and tail of a list. With this mechanism available, it is easy to recursively scan a list. As always with recursion, you will first need to define a base case: with lists, this is often the case of the empty list `'()`. Then, figure out how what should happen when recursing from this base case to the case of a one element list.

Write a function `how-many` which takes a first argument x and a second list argument l, and returns the occurrence count of x in l. Example:

```
(how-many 4 '(1 4 5 2 3 4 a b 4 4 c 4))
```

should return 5.

Write a function `duplicate` which takes a list and returns the same list where every element has been duplicated. Example:

```
(duplicate '(Do duck quacks echo?))
```

should return `'(Do Do duck duck quacks quacks echo? echo? )`

Re-write your `collatz?` function to generate a list of the successive values computed by the process.

```
(collatz-list 13)
-> '(13 40 20 10 5 16 8 4 2 1)
```