# Problem Set 5
# COMP301 Fall 2019
### 7.11.2019 17:30 - 18:45

**Read me first!** Please download the *Codes* file. In the scheme codes, you will see some hints regarding where to modify. You will use DrRacket. We have also edited the `tests.rkt` for each of them so that if you solve the problem, running `tests.rkt` should have no errors. You can also write your own program in `let`, `letrec` or `proc` by writing your code in a string and run it from the console of the respective `tests.rkt` like this:

```
(display (run ''your code here''))
```

Make sure you run the file itself to update the definitions before running your code from console. Regarding problem 2, just write the `proc` code to a text file and save it, after you make sure that it is the correct answer. As for your submission, you will submit your modified code and the text file for problem 2 in a zip folder.

**Problem 1**[1]: Extend the `let` language so that a let declaration can declare an arbitrary number of variables, using the grammar:

$$Expression ::= \text{let } \{Identifier = Expression\}^* \text{ in } Expression$$

As in Scheme's let, each of the right-hand sides are evaluated in the current environment and the body is evaluated with each new variable bound to the value of its associated right-hand side. For example:

```
let x = 30 in
   let x = -(x,1)
       y = -(x,2)
   in -(x,y)
```

evaluates to 1.

**Problem 2**[2]: In `PROC` language, procedures have only one argument, but one can get the effect of multiple argument procedures by using procedures that return other procedures. For example, one might write a code like:

```
let f = proc (x) proc (y) ...
in ((f 3) 4)
```

See, it is as if we called using two parameters as `(f 3 4)`! This trick is called *Currying* and the procedure is said to be *Curried*. Write a Curried procedure that takes two arguments and returns their sum. You do not have to modify or extend the language, just write the function. *Hint: You can write summation in Scheme language by doing* `-(x, -(0, y))`

[1] EOPL p.74 Exercise 3.16
[2] EOPL p.80 Exercise 3.20

**Problem 3**[3]: *Dynamic binding* (or *dynamic scoping*) is an alternative design for procedures, in which the procedure body is evaluated in an environment obtained by extending the environment at the point of call. For example in the code below:

```
let a = 3
in let p = proc (x) - (x, a)
   in let a = 5
      in -(a, (p, 2))
```

the `a` in the procedure body would be bound to 5, not 3. Modify the `proc` language to use *dynamic binding*.

**Problem 4**[4]: Extend the `letrec` language to allow the declaration of any number of mututally recursive unary procedures, for example:

```
letrec
  even(x) = if zero?(x) then 1 else (odd -(x,1))
  odd(x) = if zero?(x) then 0 else (even -(x,1))
in (odd 13)
```

which evaluates to 1 because 13 is an odd number.

[3]EOPL p.82 Exercise 3.28

[4]EOPL p.84-85 Exercise 3.32