# Homework 2
# COSE212, Fall 2022

Hakjoo Oh

**Due: 10/5, 23:59**

**Academic Integrity / Assignment Policy**

- All assignments must be your own work.
- Discussion with fellow students is encouraged including how to approach the problem. However, your code must be your own.
  - Discussion must be limited to general discussion and must not involve details of how to write code.
  - You must write your code by yourself and must not look at someone else's code (including ones on the web).
  - Do not allow other students to copy your code.
  - Do not post your code on the public web.
- **Violating above rules gets you 0 points for the entire HW score.**

**Problem 1** Write a higher-order function

```
dropWhile : ('a -> bool) -> 'a list -> 'a list
```

which removes elements of a list while they satisfy a predicate. For example,

```
dropWhile (fun x -> x mod 2 = 0) [2;4;7;9]
```

evaluates to `[7;9]` and

```
dropWhile (fun x-> x > 5) [1;3;7]
```

evaluates to `[1;3;7]`.

**Problem 2** Write a higher-order function

```
sigma : (int -> int) -> int -> int -> int
```

such that `sigma f a b` computes

$$\sum_{i=a}^{b} f(i).$$

For instance,

```
sigma (fun x -> x) 1 10
```

evaulates to 55 and

```
sigma (fun x -> x*x) 1 7
```

evaluates to 140.

**Problem 3** Write a higher-order function

```
forall : ('a -> bool) -> 'a list -> bool
```

which decides if all elements of a list satisfy a predicate. For example,

```
forall (fun x -> x mod 2 = 0) [1;2;3]
```

evaluates to false while

```
forall (fun x -> x > 5) [7;8;9]
```

is true.

**Problem 4** Write a function

```
uniq: 'a list -> 'a list
```

which removes duplicated elements from a given list so that the list contains unique elements. For instance,

```
uniq [5;6;5;4] = [5;6;4]
```

**Problem 5** In class, we defined the function `reverse` as follows:

```
let rec reverse l =
  match l with
  | [] -> []
  | hd::tl -> (reverse tl) @ [hd]
```

The function is slow; its time complexity is $O(n^2)$. For instance, `reverse (range 1 100000)` may not terminate quickly on typical machines. However, list reversal can be implemented efficiently with time complexity $O(n)$. Write a function

```
fastrev : 'a list -> 'a list
```

that reverses a given list with in $O(n)$. For instance, `fastrev (range 1 100000)` should produce `[100000; 99999; ...; 1]` immediately.

**Problem 6** Write a function

```
diff : aexp * string -> aexp
```

that differentiates the given algebraic expression with respect to the variable given as the second argument. The algebraic expression `aexp` is defined as follows:

```
type aexp =
 | Const of int
 | Var of string
 | Power of string * int
 | Times of aexp list
 | Sum of aexp list
```

For example, $x^2 + 2x + 1$ is represented by

```
Sum [Power ("x", 2); Times [Const 2; Var "x"]; Const 1]
```

and differentiating it (w.r.t. "x") gives $2x + 2$, which can be represented by

```
Sum [Times [Const 2; Var "x"]; Const 2]
```

Note that the representation of $2x + 2$ in `aexp` is not unique. For instance, the following also represents $2x + 2$:

```
Sum
 [Times [Const 2; Power ("x", 1)];
  Sum
   [Times [Const 0; Var "x"];
    Times [Const 2; Sum [Times [Const 1]; Times [Var "x"; Const 0]]]];
  Const 0]
```

**Problem 7** Consider the following expressions:

```
type exp = X
         | INT of int
         | ADD of exp * exp
         | SUB of exp * exp
         | MUL of exp * exp
         | DIV of exp * exp
         | SIGMA of exp * exp * exp
```

Implement a calculator for the expressions:

```
calculator : exp -> int
```

For instance,

$$\sum_{x=1}^{10}(x * x - 1)$$

is represented by

```
SIGMA(INT 1, INT 10, SUB(MUL(X, X), INT 1))
```

and evaluating it should give 375.

**Problem 8** Consider the following language:

```
type exp = V of var
         | P of var * exp
         | C of exp * exp
and var = string
```

In this language, a program is simply a variable, a procedure, or a procedure call. Write a checker function

```
check : exp -> bool
```

that checks if a given program is well-formed. A program is said to be *well-formed* if and only if the program does not contain free variables; i.e., every variable name is bound by some procedure that encompasses the variable. For example, well-formed programs are:

- `P ("a", V "a")`
- `P ("a", P ("a", V "a"))`
- `P ("a", P ("b", C (V "a", V "b")))`
- `P ("a", C (V "a", P ("b", V "a")))`

Ill-formed ones are:

- `P ("a", V "b")`
- `P ("a", C (V "a", P ("b", V "c")))`
- `P ("a", P ("b", C (V "a", V "c")))`