Tutorial 5 : Functional Programming

Do Dang Khoi

Question 1.

Let **lst** be a list of integer, write function **double(lst)** that returns the list of double of each element in **lst**
For example: double([5,7,12,-4]) returns [10,14,24,-8]

a) Use list comprehension approach?

```
def double(lst):
    return [ele * 2 for ele in lst]
```

b) Use recursive approach?

```
def double(lst):
    if len:
        return [lst[0] * 2] + double_b(lst[1:])
    else:
        return []
```

c) Use high-order function approach?

```
def double(lst):
    return list(map(lambda x: x * 2, lst))
```

Question 2.

Let **lst** be a list of a list of element, write function **flatten(lst)** that returns the list of all elements.
For example:
flatten([[1,2,3],['a','b','c'],[1.1,2.1,3.1]]) returns [1,2,3,'a','b','c',1.1,2.1,3.1]

a) Use list comprehension approach?

```
def flatten(lst):
    return [x for y in lst for x in y]
```

b) Use recursive approach?

```
def flatten(lst):
    if lst:
        return lst[0] + flatten_b(lst[1:])
    else:
        return []
```

c) Use high-order function approach?

```
def flatten(lst):
    return list(reduce(lambda x, y: x + y, lst))
```

Question 3.

Let **lst** be a list of integer and **n** be an integer, write a function **lessThan(n, lst)** that returns a list of all numbers in **lst** less than **n**.
For example:
lessThan(50, [1, 55, 6, 2]) returns [1,6,2]

a) Use list comprehension approach?

```
def lessThan(n, lst):
    return [ele for ele in lst if ele < n]
```

b) Use recursive approach?

```
def lessThan(n, lst):
    if lst:
        return lessThan_b(n, lst[1:]) if lst[0] >=n \
        else  [lst[0]] + lessThan_b(n, lst[1:])
    else:
        return []
```

c) Use high-order function approach?

```
def lessThan(n, lst):
    return list(filter(lambda ele: ele < n, lst))
```

Question 4.

Write function **compose** that can compose as many functions as you want. For example, there are three functions: **double**, **increase** and **square**. They can be called like **compose(double,increase)** or **compose(square,increase,double).**

```
def square(num):
    return num * num

def increase(num):
    return num + 1

def double(num):
    return num * 2
```

a) Use recursive approach?

```
def compose(*args):
    cps = compose(*args[:-1]) if len(args) > 2 else args[-2]
    return lambda num: cps(args[-1](num))
```

b) Use high-order function approach?

```
def compose(*args):
    def h(arg):
        return reduce(lambda x, y: y(x), reversed(args), arg)
    return h
```