National University of Singapore
School of Computing
CS1101S: Programming Methodology
Semester I, 2016/2017

Recitation 3
Higher-Order Functions

Definitions

Note: It is not necessary to memorize these definitions, or even the names of these functions. Definitions of such functions (if they are used) will be given in an Appendix for examinations. What is more important is to be able to *read* the definitions and understand how the functions work.

From Past IVLE Forum: `make_one_out_of_two`

```
// inspired by CS1101S alumnus Hoo De Lin

function make_one_out_of_two(x,y) {
    return function(m) { return m(x, y); }
}
function first(z) {
    return z(function(p, q) { return p; });
}
function second(z) {
    return z(function(p, q) { return q; });
}

// example
var my_two_values = make_one_out_of_two(42, 4711);
var my_first_value = first(my_two_values);
var my_second_value = second(my_two_values);
```

From the Lecture: `sum`

The following higher-order function was discussed in the lecture:

```
function sum(term, a, next, b) {
    if(a > b) {
        return 0;
    } else {
        return term(a) + sum(term, next(a), next, b);
    }
}
```

Problems:

1. This problem is inspired by a past IVLE Forum discussion "How to return two values within a function" (Hoo De Lin). Consider the following program:

```
function division(x, y) {
    function iter(a, result) {
        if (a < y) {
            return result;
        } else {
            return iter(a - y, result + 1);
        }
    }
    return iter(x, 0);
}

function remainder(x, y) {
    function iter(a, result) {
        if (a < y) {
            return a;
        } else {
            return iter(a - y, result + 1);
        }
    }
    return iter(x, 0);
}
```

   Use the function `one_out_of_two` to write a program `division_with_remainder` that returns both the quotient and the remainder of the division:

```
var div_result = division_with_remainder(27, 8);
first(div_result);
second(div_result);
```

2. Write a function `fast_power_iter` that raises a given base to the power of a given non-negative integer exponent, that uses the `fast_power` method from the lectures and that gives rise to an iterative process.

3. Write a function `my_sum` that computes the following sum, for $n \geq 1$:

$$1 \times 2 + 2 \times 3 + \cdots + n \times (n + 1)$$

4. Does the function `my_sum` as defined in Question 3 give rise to a recursive process or an iterative process? What is the order of growth in time and in space, using Θ notation?

5. If your function in Question 3 gives rise to a recursive process, re-write `my_sum` such that it gives rise to an iterative process. If your function in Question 3 gives rise to an iterative process, re-write `my_sum` such that it gives rise to a recursive process.

6. We can also define `my_sum` in terms of the higher-order function `sum`. Complete the definition of `my_sum` below. You cannot change the definition of `sum`; you may only call it with appropriate arguments.

   ```
   function my_sum(n) { return sum(<T1>, <T2>, <T3>, <T4>); }


   T1:


   T2:


   T3:


   T4:
   ```

7. Write an iterative version of `sum`.