National University of Singapore
School of Computing
CS1101S: Programming Methodology (JavaScript)
Semester I, 2012/2013

# Discussion Group Exercises 3

## Problems:

1. *Simpson's Rule* is a method of numerical integration. Using Simpson's Rule, the integral of a function $f$ from $a$ to $b$ is approximated as

$$\frac{h}{3}[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + y_n]$$

where $h = \frac{b-a}{n}$, for some even integer $n$, and $y_k = f(a + kh)$. (Increasing $n$ increases the accuracy of the approximation.) Define a function that takes as arguments $f$, $a$, $b$, and $n$ and returns the value of the integral, computed using the above Simpson's Rule. Use your function and attempt to integrate a cube between 0 and 1 (with $n = 100$ and $n = 1000$).

*Hint:* A sample call with $f(x) = x^3, a = 0, b = 1$, $n = 100$ should be/look like this:

```
calc_integral(function(x){ return Math.pow(x, 3); }, 0, 1, 100);
```

2. Consider the following function `sum`:

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

This function generates a linear recursion. The function can be rewritten so that the summation is performed iteratively. Show how to do this by filling in the missing expressions in the following definition:

```
function sum(term, a, next, b){
    function iter(a, result){
        if(<???>){
            return <???>;
        }else{
            return iter(<???>, <???>);
        }
    }
    return iter(<???>, <???>);
}
```

3. Write a function `g(k)` that solves the following product using the function `fold`.

$$g(k) = \prod_{x=0}^{k} (x - (x+1)^2)$$

Note that big-Pi ($\Pi$) notation used for product in the same way Sigma ($\Sigma$) notation is for `sum`. The code for `fold` is provided for you below.

```
function fold(op, f, n){
    if(n == 0){
        return f(0);
    }else{
        return op(f(n), fold(op, f, n - 1);
    }
}
```

4. (a) Show that `sum` is a special case of a still more general notion called `accumulate` that combines a collection of terms, using some general accumulation function `combiner`:

   ```
   accumulate(combiner, null_value, term, a, next, b);
   ```

   Accumulate takes as arguments the same term and range specifications as `sum`, together with a combiner function (of two arguments) that specifies how the current term is to be combined with the accumulation of the preceding terms and a null-value that specifies what base value to use when the terms run out. Write the `accumulate` function and show how `sum` can be defined as a simple call to accumulate.

   (b) If your `accumulate` function generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

5. You can obtain an even more general version of `accumulate` by introducing the notion of a filter on the terms to be combined. That is, combine only those terms derived from values in the specified range that satisfy a specified condition. The resulting `filtered_accumulate` abstraction takes the same arguments as accumulate, together with an additional predicate of one argument that specifies the filter. Write `filtered_accumulate` as a function. Show how to express the following using `filtered_accumulate`:

   (a) the sum of the squares of all the prime numbers in the interval $a$ to $b$ inclusive (assuming that you have a `is_prime` predicate already written)

   (b) the product of all the positive integers less than $n$ that are relatively prime to $n$ (i.e., all positive integers $i < n$ such that $GCD(i, n) = 1$). You can assume that you already have a function `gcd` that returns the gcd of $a$ and $b$, if you needed such a function.

6. (a) Consider the problem of representing line segments in a plane. Each segment is represented as a pair of points: a starting point and an ending point. A point can be represented as a pair of numbers: the $x$ coordinate and the $y$ coordinate. Specify a constructor `make_point` and selectors `x_point` and `y_point` that define this representation. Subsequently, define a constructor `make_segment` and selectors `start_segment` and `end_segment` that define the representation of segments in terms of points. Finally, using your selectors and constructors, define a function `midpoint_segment` that takes a line segment as argument and returns its midpoint (the point whose coordinates are the average of the coordinates of the endpoints).

   To try your functions, you may find the following function (that print a given point argument) useful:

   ```
   function print_point(p){
       display("(" + x_point(p) + "," +
               y_point(p) + ")");
   }
   ```

This is an example of why data abstraction is useful. Because `print_point` only uses the selectors for your new compound data object, it does not need to know anything about how your object is implemented. Furthermore, if you decided to modify your object's implementation, there is virtually no need to modify `print_point`.

(b) Implement a representation for a rectangle in a plane. In terms of your constructors and selectors, create functions that compute the perimeter and the area of a given rectangle.

Now implement a different representation for rectangles. Can you design your system with suitable abstraction barriers, so that the same perimeter and area functions will work using either representation?