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

Discussion Group Exercises 2

Problems:

1. Suppose we define the function:

```
function f(g) {
    return g(4);
}
```

Then we have

```
f(Math.sqrt);
2

f(function(z) { return z * (z + 1); });
20
```

What happens if we (oddly) ask the interpreter to evaluate the combination `f(f);`? Explain.

2. Draw the tree illustrating the process generated by the `cc` (count-change) function given in the lecture, in making change for 11 cents, using the denominations 50, 20, 10, 5 and 1. What are the orders of growth of the space and number of steps used by this process as the amount to be changed increases?

3. A function f is defined by the rule that $f(n) = n$ if $n < 3$ and $f(n) = f(n-1) + 2f(n-2) + 3f(n-3)$ if $n \geq 3$.

(a) Write a function that computes f by means of a recursive process.

(b) Write a function that computes f by means of an iterative process.

4. The following pattern of numbers is called Pascal's triangle.

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it.

(a) Write a function that computes elements of Pascal's triangle by means of a recursive process, i.e. the function should take two arguments, `row` and `column`, and return the element for the specified column and row. Note that only one element must be returned and NOT the entire row. E.g. calling the method with `row = 3` and `column = 2` should return `2`. Likewise calling the method with `row = 5` and `column = 3` should return `6`.

(b) Convert the function you wrote as a solution to (a) into a iterative process.

5. Write an iterative, logarithmic-time version of `together_copies_of` given in the lecture.

   There are more discussion group exercises in the next section.

Higher-Order Functions

One of the things that makes JavaScript different from other common programming languages is the ability to operate with *higher-order functions*, namely, functions that manipulate and generate other functions. The problem set will give you extensive practice with higher-order functions and simple compound data types, in the context of a language for graphing two-dimensional curves and other shapes.

In the problem set, we will be using many functions which may be applied to many different types of arguments and may return different types of values. To keep track of this, it will be helpful to have some simple notation to describe types of JavaScript values.

Two basic types of values are JS-Num and JS-Bool. JS-Num are the JavaScript numbers such as 3, -4.2, 6.931479453e89. JS-Bool are the truth values `true,false`. The function `Math.sqrt` may be applied to a JS-Num and will return another JS-Num. We indicate this with the notation:

$$\texttt{Math.sqrt} : \text{JS-Num} \rightarrow \text{JS-Num}$$

If `f` and `g` are functions of type JS-Num → JS-Num, then we may *compose* them:

```
function compose(f, g){
   return function(x){
              return f(g(x));
          }
}
```

Thus, for example `compose(Math.sqrt, Math.log)` is the function of type JS-Num → JS-Num that returns the square root of the logarithm of its argument, while `compose(Math.log, Math.sqrt)` returns the logarithm of the square root of its argument:

```
Math.log(2);
// Value: 0.6931471805599453

(compose(Math.sqrt, Math.log))(2);
// Value: 0.8325546111576977

(compose(Math.log, Math.sqrt))(2);
// Value: 0.3465735902799727
```

As we have used it above, the function `compose` takes as arguments two functions of type F = JS-Num → JS-Num, and returns another such function. We indicate this with the notation:

$$\texttt{compose} : (F, F) \rightarrow F$$

Just as squaring a number multiplies the number by itself, `thrice` of a function composes the function three times. That is, `thrice(f)(n)` will return the same number as `f(f(f(n)))`:

```
function thrice(f) {
    return compose(compose(f, f), f);
}
```

```
thrice(Math.sqrt)(6561);
// Value: 3

Math.sqrt(Math.sqrt(Math.sqrt(6561)));
// Value: 3
```

As used above, `thrice` is of type $(F \to F)$. That is, it takes as input a function from numbers to numbers and returns the same kind of function. But `thrice` will actually work for other kinds of input functions. It is enough for the input function to have a type of the form $T \to T$, where T may be any type. So more generally, we can write

$$\texttt{thrice} : (T \to T) \to (T \to T)$$

Composition, like multiplication, may be iterated. Consider the following:

```
function identity(x) { return x; }

function repeated(f, n) {
    if (n === 0) {
        return identity;
    } else {
        return compose(f, repeated(f, n - 1));
    }
}

repeated(Math.sin, 5)(3.1);
// Value: 0.041532801333692235

Math.sin(Math.sin(Math.sin(Math.sin(Math.sin(3.1)))));
// Value: 0.041532801333692235
```

$$\texttt{repeated} : ((T \to T), \text{JS-Nonneg-Int}) \to (T \to T)$$

6. The type of `thrice` is of the form $(T' \to T')$ (where T' happens to equal $(T \to T)$), so we can legitimately use `thrice` as an input to `thrice`!

   For what value of `n` will `thrice(thrice)(f)(0)` return the same value[1] as `repeated(f, n)(0)`?

   See if you can now predict what will happen when the following expressions are evaluated. Briefly explain what goes on in each case.

   Note: Function `sqr` and `add1` are defined as follows:

```
function sqr(x) {
    return x * x;
}
function add1(x) {
    return x + 1;
}
```

   (a) `thrice(thrice)(add1)(6)`

[1]"Sameness" of function values is a sticky issue which we don't want to get into here. We can avoid it by assuming that `f` is bound to a value of type F, so evaluation of `thrice(thrice)(f)(0)` will return a number.

(b) `thrice(thrice)(identity)(compose)`

(c) `thrice(thrice)(sqr)(1)`

(d) `thrice(thrice)(sqr)(2)`