National University of Singapore
School of Computing
CS1101S: Programming Methodology
Semester I, 2024/2025

S4-in-class
Higher-order Functions

In this in-class sheet, we will refer to the notation and definitions specified in the studio S4 sheet. We indicate a function `f` can be applied to a *Number* and will return another *Number* with the following notation:

$$\texttt{f} : \textit{Number} \rightarrow \text{Number}$$

We call the type of this function *Number-Transformation*:

$$\textit{Number-Transformation} := \textit{Number} \rightarrow \textit{Number}$$

and thus we can also say

$$\texttt{f} : \textit{Number-Transformation}$$

Problems:

1. As we have seen in question 3 of the studio sheet, the function `compose` takes as arguments two functions of type *Number-Transformation*, and returns another such function. We indicate this with the notation:

   $$\texttt{compose} : (\textit{Number-Transformation}, \textit{Number-Transformation}) \rightarrow \textit{Number-Transformation}$$

   Just as squaring a number multiplies the number by itself, applying `thrice` to 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);
   }
   ```

   What is the result of evaluating `thrice(math_sqrt)(256);`?

2. As used above, `thrice` is of type *Number-Transformation* → *Number-Transformation*, which, upon expansion, is:

   $$(\textit{Number} \rightarrow \textit{Number}) \rightarrow (\textit{Number} \rightarrow \textit{Number})$$

   That is, it takes as input a function from numbers to numbers and returns the same kind of function. (You may call `thrice` a `Number-Transformation-Transformation`.) But `thrice` will actually work on other kinds of transformations, not just `Number-Transformation`s: It is enough for the input function to have a type of the form $T \rightarrow T$, where T may be *any* type. So more generally, we can write

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

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

```
function repeated(f, n) {
    return n === 0
        ? x => x
        : 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
```

We can write

$$\texttt{repeated}: ((T \to T), \mathit{Number}) \to (T \to T)$$

Implement `thrice` using `repeated`.

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 as `repeated(f, n)(x)`? In other words: What is the result of the following program?

```
thrice(thrice)(x => x + 1)(0);
```

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

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

   ```
   const square = x => x * x;
   const add1 = x => x + 1;
   ```

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

   (b) `((thrice(thrice))(x => x))(compose);`

   (c) `((thrice(thrice))(square))(1);`

   (d) `((thrice(thrice))(square))(2);`