The lectures explained a function for function composition:

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

Now `compose(f(compose(f,f)))` returns a function that applies `f` three times to its argument. The following function generalizes this concept of repeated application.

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

For example, repeated application of an increment function implements addition (of course in a rather inefficient way):

```javascript
function increment(x) { return x + 1; }
function plus(x, y) {
    return (repeated(increment, y))(x);
}
```

As an alternative definition of the `fold` function presented in Lecture 6, consider the following function `together_copies_of`.

```javascript
function together_copies_of(f, n, a) {
    return (repeated(function(x) { return f(a, x); }, n - 1))(a);
}
```

To see `together_copies_of` in action, consider the following alternative definition of `power`:

```javascript
function power(b, e) {
    return together_copies_of(function (x, y) { return x * y; }, e, b);
}
```