WEEK 4 STUDIO

STUDIO 3 QUESITION 2

Ben Bitdiddle's claim actually depends on the base of the logarithm!

THIS WEEK'S AGENDA

- Coin Change
- Higher Order Functions
- Studio

COIN CHANGE

How many ways can I change x amount of money given n kinds of coins?

- *Amount: 1.50*
- *Coins: unlimited amount of 100, 50, 20, 10, and 5 cent coins*

COIN CHANGE

- Base case:
  - 0 amount of money left — 1
  - 0 kinds of coins — 0
  - < 0 amount of money left — 0
- Smaller problem:
  - Number of ways to change money **using first kind of coin**
  - Number of ways to change money **without using first kind of coin**

COIN CHANGE

```
function count_change(amount) {
    return cc(amount, 5);
}
function cc(amount, kinds_of_coins) {
    return amount === 0
           ? 1
           : amount < 0 ||
             kinds_of_coins === 0
             ? 0
             : cc(amount, kinds_of_coins - 1)
               +
               cc(amount - first_denomination(
                                kinds_of_coins),
                  kinds_of_coins);
}
```

COIN CHANGE

```
function first_denomination(kinds_of_coins) {
    return kinds_of_coins === 1 ? 5 :
           kinds_of_coins === 2 ? 10 :
           kinds_of_coins === 3 ? 20 :
           kinds_of_coins === 4 ? 50 :
           kinds_of_coins === 5 ? 100 : 0;
}
```

BEFORE HIGHER ORDER FUNCTIONS

- Variable scoping
- Function definition expressions

VARIABLE SCOPING

- **Pre-declared** functions or constants are visible anywhere
- **User declared** functions or constants are only visible within the block that they are declared (i.e. the closest surrouding curly braces)
- **Function parameters** are visible only to the body of the functions that they belong

> ***Scoping rule:*** *name occurrence refers to the closest surrounding declaration*

VARIABLE SCOPING

```
const x = 5;

function f(x) {
    return x;
}

f(3);
```

What is the result of this program?

Ans: 3

VARIABLE SCOPING

```
const x = 5;

function f(y) {
    const x = 10;
    function g(y) {
        return x;
    }
    return g(x);
}

f(x);
```

What is the result of this program?

Ans: 10

VARIABLE SCOPING

```
const x = 5;

function f2(y) {
    const a = 10;
    function g(y) {
        return x;
    }
    return g(a);
}

f2(x);
```

What is the result of this program?

Ans: 5

FUNCTION DEFINITION EXPRESSIONS

An expression that defines a function, but does not assign a name to the function (other name: arrow functions)

```
x => x + 1;
```

Question: Can you still assign a name to the arrow function?

FUNCTION DEFINITION EXPRESSIONS

Yes, these functions can still be assigned a name

```
const addone = x => x + 1;
addone(1); // returns 2
```

- Why?
  - More concise way of defining functions
  - Often used for short, one-lined functions where names are not needed
  - Higher order functions

HIGHER ORDER FUNCTIONS

- What?
  - Functions can be passed into other functions as arguments
  - Functions can be returned by other functions as return values

HIGHER ORDER FUNCTIONS

```
/*
Consider a restricted form of Source in which functions are
allowed to have at most one parameter. Rewrite the following
function definition under this restriction:
*/

function myfunc(a, b, c) {
    return a * b + c;
}

/*
With this new function definition, how do you rewrite the
function call myfunc(3, 2, 1)?
*/
```

HIGHER ORDER FUNCTIONS

```
function myfunc(a) {
    return b => c => a * b + c;
}

((myfunc(3))(2))(1);
```

HIGHER ORDER FUNCTIONS

```
function plus_one(x) {
    return x + 1;
}
function trans(func) {
    return x => 2 * func(x * 2);
}
function twice(func) {
    return x => func(func(x));
}
const thrice = f => f(f(f(x)));

// Evaluate the following
((twice(trans(plus_one))))(1);
((twice(trans))(plus_one))(1);
((thrice(trans))(plus_one))(1);
```

(twice(trans(plus_one)))(1): 26

((twice(trans))(plus_one))(1): 20

((thrice(trans))(plus_one))(1): 72

HIGHER ORDER FUNCTIONS

- Why?
  - Might not see the use now, but later when **map**, **accumulate** and **filter** comes it, they'll be really handy.
  - **Map**, **accumulate**, **filter** are really useful functions for data manipulation within data structures.

STUDIO SHEET

THRICE THRICE THRICE

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

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

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

THRICE THRICE THRICE

1. `thrice(thrice)(f)` ≡ `thrice(thrice(thrice(f)))`
2. `thrice(thrice)(f)` ≡ `thrice(thrice(x => f`^{3}`(x)))`
3. `thrice(thrice)(f)` ≡ `thrice(x => f`^{9}`(x))`
4. `thrice(thrice)(f)` ≡ `x => f`^{27}`(x)`

ATTENDANCE