Debugging

Call Stack

In computer science, a call stack is a stack data structure that stores information about the active subroutines of a computer program. This kind of stack is also known as an execution stack, program stack, control stack, run-time stack, or machine stack, and is often shortened to just "the stack".

Debugging

Debugging is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system.

Breakpoint

In software development, a breakpoint is an intentional stopping or pausing place in a program, put in place for debugging purposes.

Steps

Step into

Step out

Step over

List Operations

List Operation

- Transformation
- Exclusion
- Composition
- Iteration

Transformation

```
function doubleIt(x) {
    return x * 2;
}

function transform(arr, fn) {
    var list = [];
    for (var i = 0; i < arr.length; i++) {
        list[i] = fn(arr[i]);
    }

    return list; // immutable
}

let result = transform([1, 2, 3], doubleIt);
console.log(result);
```

Transformation + curry

```
function getIncreaseFunction(val) {
    return function(x) {
        return x + val;
    };
}

function transform(arr, fn) {
    var list = [];
    for (var i = 0; i < arr.length; i++) {
        list[i] = fn(arr[i]);
    }

    return list;
}

let changeFunction = getIncreaseFunction(4);

let result = transform([1, 2, 3], changeFunction);
console.log(result);
```

Map

```
function doubleIt(x) {
    return x * 2;
}

let result = [1, 2, 3].map(doubleIt);
console.log(result);
```

Map + curry

```
function getIncreaseFunction(val) {
    return function(x) {
        return x + val;
    };
}

let changeFunction = getIncreaseFunction(4);

let result = [1, 2, 3].map(changeFunction);
console.log(result);
```

Exclusion

```
function isOdd(x) {
    return x % 2 === 1;
}

function exclude(arr, fn) {
    var list = [];
    for (var i = 0; i < arr.length; i++) {
        if (fn(arr[i])) {
            list.push(arr[i]);
        }
    }

    return list;
}

let result = exclude([1, 2, 3, 4, 5], isOdd);
console.log(result);
```

Exclusion

```
function isOdd(x) {
    return x % 2 === 1;
}

let result = [1, 2, 3, 4, 5].filter(isOdd);
console.log(result);
```

Composition

```
function mult(x, y) {
    return x * y;
}

function compose(arr, fn, initial) {
    var result = initial;
    for (var i = 0; i < arr.length; i++) {
        result = fn(result, arr[i]);
    }

    return result;
}

let result = compose(
                    [1, 2, 3, 4],
                    mult,
                    1
                    );

console.log(result);
```

Iteration

```
function logValue(x) {
    console.log(x);
}

function iterate(arr, fn) {
    for (var i = 0; i < arr.length; i++) {
        fn(arr[i]);
    }
}

iterate([1, 2, 3, 4], logValue);
```

Iteration

```
function logValue(x) {
    console.log(x);
}

[1, 2, 3, 4].forEach(logValue);
```

Exercise 1

```
function foo() {
    return 42;
}

function bar() {
    return 10;
}

function add(x, y) {
    return x + y;
}

function add2(fn1, fn2) {
    return add(fn1(), fn2());
}

console.log(add(foo(), bar()));
console.log(add2(foo, bar));
```

Exercise 2

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

function add(x, y) {
    return x + y;
}

function add2(fn1, fn2) {
    return add(fn1(), fn2());
}

console.log(add(foo(10)(), foo(42)()));
console.log(add2(foo(10), foo(42)));
```

Exercise 3 (1/2) - iteration

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

function add(x, y) {
    return x + y;
}

function add2(fn1, fn2) {
    return add(fn1(), fn2());
}
```

Exercise 3 (2/2) - iteration

```
function addn(arr) {
    var sum = 0;
    for (var i = 0; i < arr.length; i++) {
        sum = add2(arr[i], foo(sum));
    }

    return sum;
}

let result = addn([foo(1), foo(2), foo(3), foo(4)]);
console.log(result);
```

Exercise 4 (1/2) - recursion

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

function add(x, y) {
    return x + y;
}

function add2(fn1, fn2) {
    return add(fn1(), fn2());
}
```

Exercise 4 (2/2) - recursion

```
function addn(arr) {
    if (arr.length === 0) {
        return add2(foo(0), foo(0));
    }
    if (arr.length === 1) {
        return add2(arr[0], foo(0));
    }
    if (arr.length === 2) {
        return add2(arr[0], arr[1]);
    }
    return addn(
        [
            function() {
                return add2(arr[0], arr[1]);
            }
        ].concat(arr.slice(2))
    );
}

let result = addn([foo(1), foo(2), foo(3), foo(4)]);
console.log(result);
```

Exercise 5 (1/2) - reduce

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

function add(x, y) {
    return x + y;
}

function add2(fn1, fn2) {
    return add(fn1(), fn2());
}
```

Exercise 5 (2/2)- reduce

```
function addn(arr) {
    return arr.slice(1).reduce(function(prev, cur) {
        return function() {
            return add2(prev, cur);
        };
    }, arr[0])();
}

let result = addn([foo(1), foo(2), foo(3), foo(4)]);
console.log(result);
```

Exercise 6 (1/2) - map + reduce

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

function add(x, y) {
    return x + y;
}

function add2(fn1, fn2) {
    return add(fn1(), fn2());
}
```

Exercise 6 (2/2) - map + reduce

```
function addn(arr) {
return arr.map(foo).reduce(function(prev, cur) {
        return function() {
            return add2(prev, cur);
        };
    }, foo(0))();
}

let result = addn([1, 2, 3, 4]);
console.log(result);
```