Save this in an **.html** file.

```
<html><body><pre><script>
function log(arg) {
    document.writeln(arg);
}
function identity(x) {
    return x;
}
log(identity(3));
</script></pre></body></html>
```

Fun With Functions

Douglas Crockford

Write an **identity** function that takes an argument and returns that argument.

identity(3) // 3

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


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

```
<html><body><pre><script>
function log(arg) {
    document.writeln(arg);
}
function identity(x) {
    return x;
}
log(identity(3));
</script></pre></body></html>
```

The Rules

1. If you have a question, you must ask it.
2. If you need more time, you must say so.
3. I won’t debug your stuff.

Quiz

```
function funky(o) {
    o = null;
}
var x = [];
funky(x);
log(x);
```

A.null
B.[]
C.undefined
D.throw

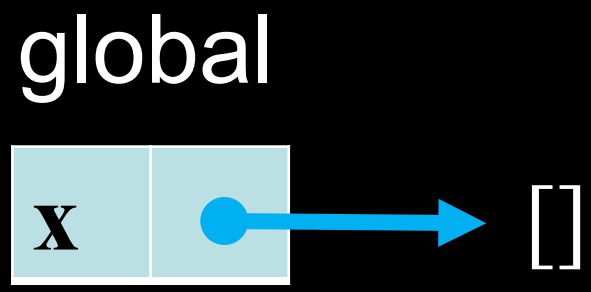
What is x?

```
function funky(o) {
    o = null;
}
var x = [];
funky(x);
log(x);
```

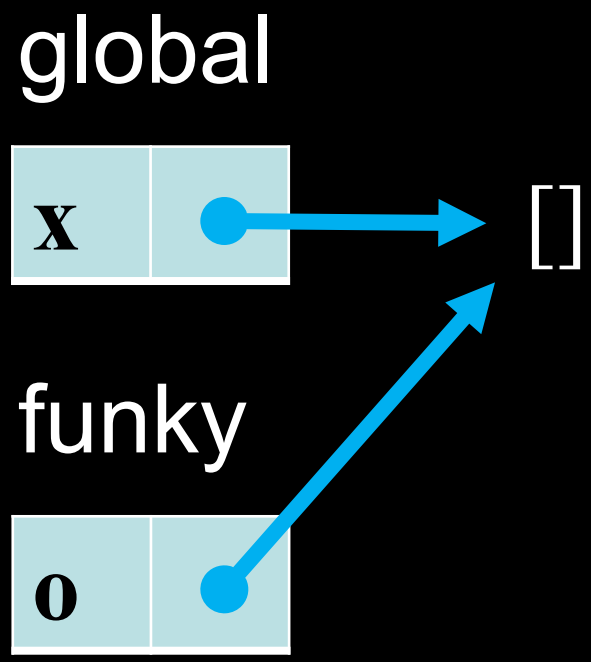
A.null
B.[]
C.undefined
D.throw

What is x?

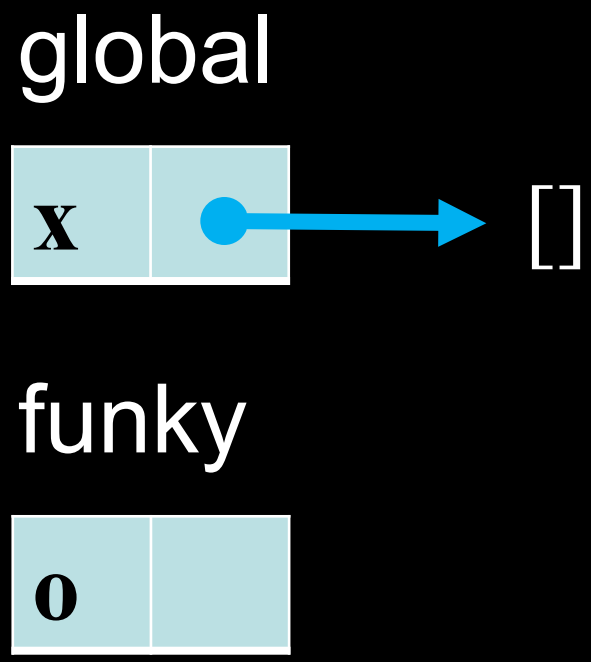
```
function funky(o) {
    o = null;
}
var x = [];
funky(x);
log(x);
```



```
function funky(o) {
    o = null;
}
var x = [];
funky(x);
log(x);
```



```
function funky(o) {
    o = null;
}
var x = [];
funky(x);
log(x);
```



```
function swap(a, b) {
    var temp = a;
    a = b;
    b = temp;
}
var x = 1;
var y = 2;
swap(x, y);
log(x);
```

A.1
B.2
C.undefined
D.throw

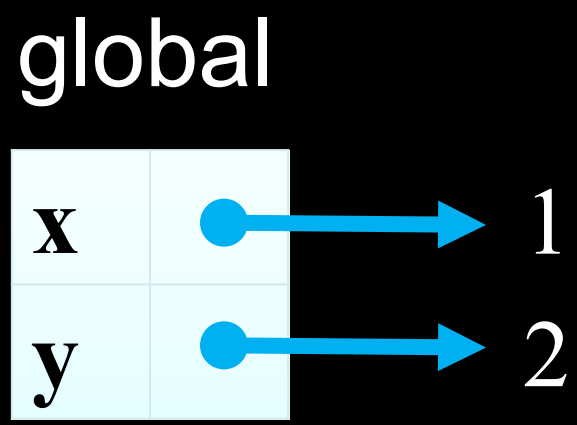
What is x?

```
function swap(a, b) {
    var temp = a;
    a = b;
    b = temp;
}
var x = 1;
var y = 2;
swap(x, y);
log(x);
```

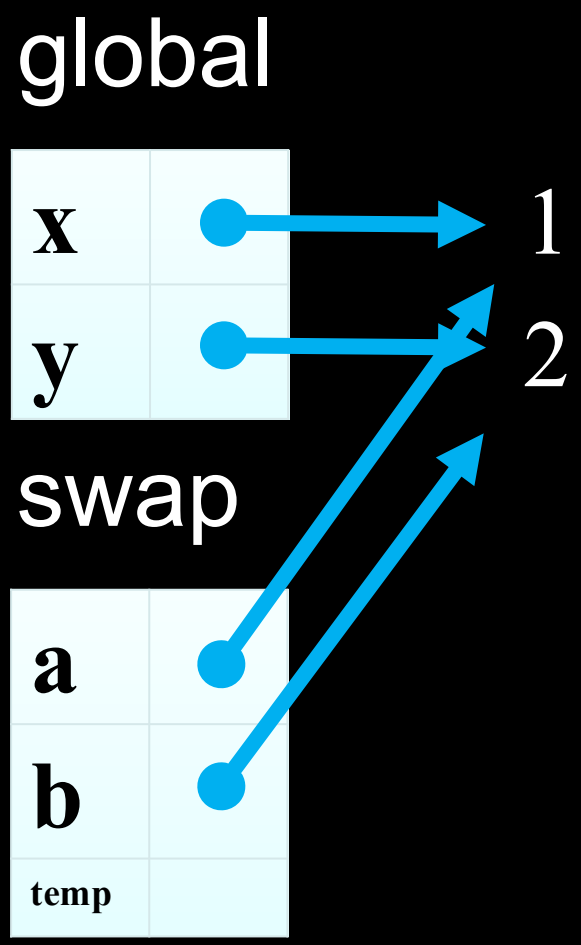
A.1
B.2
C.undefined
D.throw

What is x?

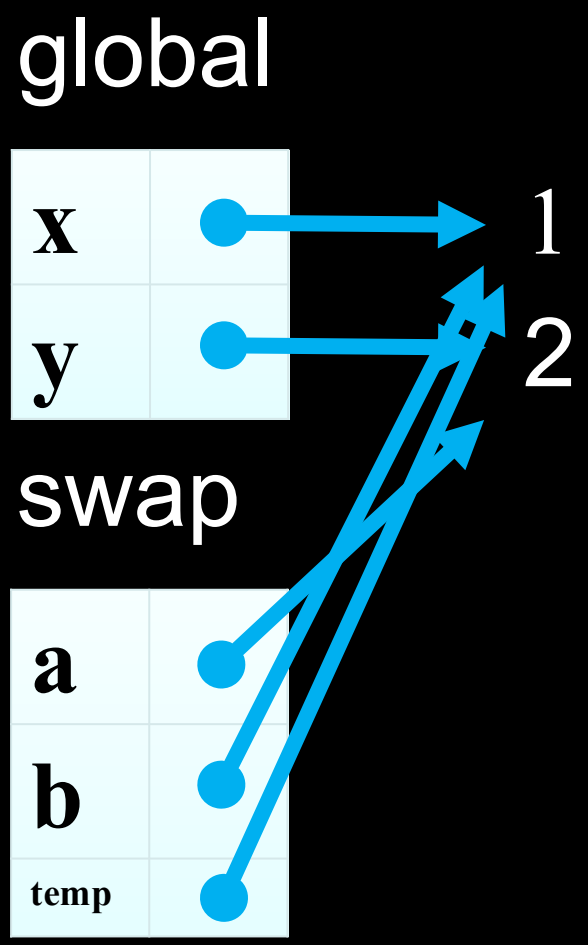
```
function swap(a, b) {
    var temp = a;
    a = b;
    b = temp;
}
var x = 1;
var y = 2;
swap(x, y);
log(x);
```



```
function swap(a, b) {
    var temp = a;
    a = b;
    b = temp;
}
var x = 1;
var y = 2;
swap(x, y);
log(x);
```



```
function swap(a, b) {
    var temp = a;
    a = b;
    b = temp;
}
var x = 1;
var y = 2;
swap(x, y);
log(x);
```



Write three binary functions, **add**, **sub**, and **mul**, that take two numbers and return their sum, difference, and product.

```
add(3, 4)    //  7

sub(3, 4)    // -1

mul(3, 4)    // 12
```

```
function add(first, second) {
    return first + second;
}

function sub(first, second) {
    return first - second;
}

function mul(first, second) {
    return first * second;
}
```

Write a function **identityf** that takes an argument and returns a function that returns that argument.

```
var three = identityf(3);
three()    // 3
```

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

Write a function **addf** that adds from two invocations.

```
addf(3)(4)    // 7
```

```
function addf(first) {
    return function (second) {
        return first + second;
    };
}
```

Write a function **curry** that takes a binary function and an argument, and returns a function that can take a second argument.

```
var add3 = curry(add, 3);
add3(4)              // 7

curry(mul, 5)(6)     // 30
```

```
function curry(binary, first) {
    return function (second) {
        return binary(first, second);
    };
}
```

```
function curry(binary, first) {
    return function (second) {
        return binary(first, second);
    };
}
```

currying

schönfinkelisation

```
function curry(func) {
    var slice = Array.prototype.slice;
    var args = slice.call(arguments, 1);
    return function () {
        return func.apply(
            null,
            args.concat(slice.call(arguments, 0))
        );
    };
}
```

```
function curry(func) {
    var slice = Array.prototype.slice;
    var args = slice.call(arguments, 1);
    return function () {
        return func.apply(
            null,
            args.concat(slice.call(arguments, 0))
        );
    };
}

function curry(func, ...first) {
    return function (...second) {
        return func(...first, ...second);
    };
}
```

Write a function **curryr** that takes a binary function and a second argument, and returns a function that can take a first argument.

```
var sub3 = curryr(sub, 3);
sub3(11)     // 8
sub3(3)      // 0
```

```
function curryr(binary, second) {
    return function (first) {
        return binary(first, second);
    };
}
```

Write a function **liftf** that takes a binary function, and makes it callable with two invocations.

```
var addf = liftf(add);
addf(3)(4)          // 7
liftf(mul)(5)(6)    // 30
```

```
function liftf(binary) {
    return function (first) {
        return function (second) {
            return binary(first, second);
        };
    };
}
```

```
function liftf(binary) {
    return function (first) {
        return function (second) {
            return binary(first, second);
        };
    };
}

function liftf(binary) {
    return function (first) {
        return curry(binary, first);
    };
}
```

Without writing any new functions, show four ways to create the **inc** function.

```
var inc = _ _ _ ;
inc(5)              // 6
inc(inc(5))         // 7
```

1. inc = addf(1);

2. inc = curry(add, 1);

3. inc = curryr(add, 1);

4. inc = liftf(add)(1);

Write a function **twice** that takes a binary function and returns a unary function that passes its argument to the binary function twice.

```
add(11, 11)   // 22
var doubl = twice(add);
doubl(11)     // 22
var square = twice(mul);
square(11)    // 121
```

```
function twice(binary) {
    return function (a) {
        return binary(a, a);
    };
}
```

Write **reverse**, a function that reverses the arguments of a binary function.

```
var bus = reverse(sub);
bus(3, 2)    // -1
```

```
function reverse(binary) {
    return function (first, second) {
        return binary(second, first);
    };
}

function reverse(func) {
    return function (...args) {
        return func(...args.reverse());
    };
}
```

Write a function **composeu** that takes two unary functions and returns a unary function that calls them both.

```
composeu(doubl, square)(5)    // 100
```

```
function composeu(f, g) {
    return function (a) {
        return g(f(a));
    };
}
```

Write a function **composeb** that takes two binary functions and returns a function that calls them both.

composeb(add, mul)(2, 3, 7) // 35

```
function composeb(f, g) {
    return function (a, b, c) {
        return g(f(a, b), c);
    };
}
```

Write a **limit** function that allows a binary function to be called a limited number of times.

```
var add_ltd = limit(add, 1);
add_ltd(3, 4)    // 7
add_ltd(3, 5)    // undefined
```

```
function limit(binary, count) {
    return function (a, b) {
        if (count >= 1) {
            count -= 1;
            return binary(a, b);
        }
        return undefined;
    };
}
```

Generator

A function that returns a value from a sequence.

Generator Factory

A function that returns a generator.

Write a **from** factory that produces a generator that will produce a series of values.

```
var gen = from(0);
gen()    // 0
gen()    // 1
gen()    // 2
```

```
function from(start) {
    return function () {
        var next = start;
        start += 1;
        return next;
    };
}
```

Write a **to** factory that takes a generator and an end value, and returns a generator that will produce numbers up to that limit.

```
var gen = to(from(3), 5);
gen()    // 3
gen()    // 4
gen()    // undefined
```

```
function to(gen, end) {
    return function () {
        var value = gen();
        if (value < end) {
            return value;
        }
    };
}
```

Write a **fromTo** factory that produces a generator that will produce values in a range.

```
var gen = fromTo(0, 3);
gen()   // 0
gen()   // 1
gen()   // 2
gen()   // undefined
```

```
function fromTo(start, end) {
    return to(
        from(start),
        end
    );
}
```

Write an **element** factory that takes an array and a generator and returns a generator that will produce elements from the array.

```
var gen = element(
    ["a", "b", "c", "d"],
    fromTo(1, 3)
);
gen()    // "b"
gen()    // "c"
gen()    // undefined
```

```
function element(array, gen) {
    return function () {
        var index = gen();
        if (index !== undefined) {
            return array[index];
        }
    };
}
```

Modify the **element** factory so that the generator argument is optional. If a generator is not provided, then each of the elements of the array will be produced.

```
var gen = element(
    ["a", "b", "c", "d"]
);
gen()    // "a"
gen()    // "b"
gen()    // "c"
gen()    // "d"
gen()    // undefined
```

```
function element(array, gen) {
    if (gen === undefined) {
        gen = fromTo(
            0,
            array.length
        );
    }
    return function () {
        var index = gen();
        if (index !== undefined) {
            return array[index];
        }
    };
}
```

Write a **collect** generator that takes a generator and an array and produces a function that will collect the results in the array.

```
var array = [];
var gen = collect(fromTo(0, 2), array);
gen()    // 0
gen()    // 1
gen()    // undefined
array    // [0, 1]
```

```
function collect(gen, array) {
    return function () {
        var value = gen();
        if (value !== undefined) {
            array.push(value);
        }
        return value;
    };
}
```

Write a **filter** factory that takes a generator and a predicate and produces a generator that produces only the values approved by the predicate.

```
var gen = filter(
    fromTo(0, 5),
    function third(value) {
        return (value % 3) === 0;
    }
);
gen()    // 0
gen()    // 3
gen()    // undefined
```

```
function filter(gen, predicate) {
    return function () {
        var value;
        do {
            value = gen();
        } while (
            value !== undefined &&
            !predicate(value)
        );
        return value;
    };
}
```

```
function filter(gen, predicate) {
    return function recur() {
        var value = gen();
        if (
            value === undefined ||
            predicate(value)
        ) {
            return value;
        }
        return recur();
    };
}
```

Write a **concat** factory that takes two generators and produces a generator that combines the sequences.

```
var gen = concat(
    fromTo(0, 3),
    fromTo(0, 2)
);
gen()    // 0
gen()    // 1
gen()    // 2
gen()    // 0
gen()    // 1
gen()    // undefined
```

```
function concat(gen1, gen2) {
    var gen = gen1;
    return function () {
        var value = gen();
        if (value !== undefined) {
            return value;
        }
        gen = gen2;
        return gen();
    };
}
```

```
function concat(...gens) {
    var next = element(gens);
    var gen = next();
    return function recur() {
        var value = gen();
        if (value === undefined) {
            gen = next();
            if (gen !== undefined) {
                return recur();
            }
        }
        return value;
    };
}
```

Make a function **gensymf** that makes a function that generates unique symbols.

```
var geng = gensymf("G");
var genh = gensymf("H");
geng()      // "G1"
genh()      // "H1"
geng()      // "G2"
genh()      // "H2"
```

```
function gensymf(prefix) {
    var gen = from(1);
    return function () {
        return prefix + gen();
    };
}
```

Write a function **gensymff** that takes a seed and returns a **gensymf**.

```
var gensymf = gensymff(1);
var geng = gensymf("G");
var genh = gensymf("H");
geng()      // "G1"
genh()      // "H1"
geng()      // "G2"
genh()      // "H2"
```

```
function gensymff(seed) {
    return function (prefix) {
        var gen = from(seed);
        return function () {
            return prefix + gen();
        };
    };
}
```

Make a function **fibonaccif** that returns a generator that will return the next fibonacci number.

```
var fib = fibonaccif(0, 1);
fib()   // 0
fib()   // 1
fib()   // 1
fib()   // 2
fib()   // 3
fib()   // 5
```

```
function fibonaccif(a, b) {
    var i = 0;
    return function () {
        var next;
        switch (i) {
        case 0:
            i = 1;
            return a;
        case 1:
            i = 2;
            return b;
        default:
            next = a + b;
            a = b;
            b = next;
            return next;
        }
    };
}
```

```
function fibonaccif(a, b) {
    return function () {
        var next = a;
        a = b;
        b += next;
        return next;
    };
}
```

```
var single = composeu(
    identityf,
    curryr(limit, 1)
);
function fibonaccif(a, b) {
    return concat(
        concat(single(a),single(b)),
        function fibonacci() {
            var next = a + b;
            a = b;
            b = next;
            return next;
        }
    );
}
```

```
function fibonaccif(a, b) {
    return concat(
        element([a, b]),
        function fibonacci() {
            var next = a + b;
            a = b;
            b = next;
            return next;
        }
    );
}
```

Write a **counter** constructor that returns an object containing two functions that implement an up/down counter, hiding the counter.

```
var object = counter();
var up = object.up;
var down = object.down;
up()     // 1
down()   // 0
down()   // -1
up()     // 0
```

```
function counter() {
    var value = 0;
    return {
        up: function () {
            value += 1;
            return value;
        },
        down: function () {
            value -= 1;
            return value;
        }
    };
}
```

Make a **revocable** constructor that takes a binary function, and returns an object containing an **`invoke`** function that can invoke the binary function, and a **`revoke`** function that disables the **`invoke`** function.

```
var rev = revocable(add);
var add_rev = rev.invoke;
add_rev(3, 4);    // 7
rev.revoke();
add_rev(5, 7);    // undefined
```

```
function revocable(binary) {
    return {
        invoke: function (first, second) {
            if (binary !== undefined) {
                return binary(
                    first,
                    second
                );
            }
        },
        revoke: function () {
            binary = undefined;
        }
    };
}
```

Write a constructor **m** that takes a value and an optional source string and returns them in an object.

```
JSON.stringify(m(1))
// {"value": 1, "source": "1"}
JSON.stringify(m(Math.PI, "pi"))
// {"value": 3.14159…, "source": "pi"}
```

```
function m(value, source) {
    return {
        value: value,
        source: (typeof source === "string")
            ? source
            : String(value)
    };
}
```

Write a function **addm** that takes two **m** objects and returns an **m** object.

```
JSON.stringify(addm(m(3), m(4)))
// {"value": 7, "source": "(3+4)"}
JSON.stringify(addm(m(1), m(Math.PI, "pi")))
// {"value": 4.14159…, "source": "(1+pi)"}
```

```
function addm(a, b) {
    return m(
        a.value + b.value,
        "(" + a.source + "+" +
            b.source + ")"
    );
}
```

Write a function **liftm** that takes a binary function and a string and returns a function that acts on **m** objects.

```
var addm = liftm(add, "+");
JSON.stringify(addm(m(3), m(4)))
  // {"value": 7, "source": "(3+4)"}
var mulm = liftm(mul, "*");
JSON.stringify(mulm(m(3), m(4)))
  // {"value": 12, "source": "(3*4)"}
```

```
function liftm(binary, op) {
    return function (a, b) {
        return m(
            binary(a.value, b.value),
            "(" + a.source + op
                + b.source + ")"
        );
    };
}
```

Modify function **liftm** so that the functions it produces can accept arguments that are either numbers or **m** objects.

```
var addm = liftm(add, "+");

JSON.stringify(addm(3, 4))

// {"value": 7, "source": "(3+4)"}
```

```
function liftm(binary, op) {
    return function (a, b) {
        if (typeof a === "number") {
            a = m(a);
        }
        if (typeof b === "number") {
            b = m(b);
        }
        return m(
            binary(a.value, b.value),
            "(" + a.source + op +
                b.source + ")"
        );
    };
}
```

Write a **repeat** function that takes a generator and calls it until it returns **undefined**.

```
var array = [];
repeat(collect(fromTo(0, 4), array));
log(array);    // 0, 1, 2, 3
```

```
function repeat(gen) {
    var value;
    do {
        value = gen();
    } while (value !== undefined);
}
```

```
function repeat(gen) {
    var value;
    do {
        value = gen();
    } while (value !== undefined);
}

function repeat(gen) {
    if (gen() !== undefined) {
        return repeat(gen);
    }
}
```

Write a **map** function that takes an array and a unary function, and returns an array containing the result of passing each element to the unary function. Use the **repeat** function.

```
map([2, 1, 0], inc)    // [3, 2, 1]
```

```
function map(array, unary) {
    var ele = element(array);
    var result = [];
    repeat(collect(function () {
        var value = ele();
        if (value !== undefined) {
            return unary(value);
        }
    }, result));
    return result;
}
```

Write a **reduce** function that takes an array and a binary function, and returns a single value.
Use the **repeat** function.

```
reduce([], add)          // undefined
reduce([2], add)          // 2
reduce([2, 1, 0], add)    // 3
```

```
function reduce(array, binary)  {
    var ele = element(array);
    var result;
    repeat(function () {
        var value = ele();
        if (value !== undefined) {
            result = (result === undefined)
                ? value
                : binary(result, value);
        }
        return value;
    });
    return result;
}
```

Write a function **exp** that evaluates simple array expressions.

```
var sae = [mul, 5, 11];
exp(sae)    // 55
exp(42)     // 42
```

```
function exp(value) {
    return (Array.isArray(value))
        ? value[0](
            value[1],
            value[2]
        )
        : value;
}
```

Modify **exp** to evaluate nested array expressions.

```
var nae = [
    Math.sqrt,
    [
        add,
        [square, 3],
        [square, 4]
    ]
];
exp(nae)    // 5
```

```
function exp(value) {
    return (Array.isArray(value))
        ? value[0](
            exp(value[1]),
            exp(value[2])
        )
        : value;
}

// recursion: a function calls itself
```

Write a function **addg** that adds from many invocations, until it sees an empty invocation.

```
addg()              // undefined
addg(2)()           // 2
addg(2)(7)()        // 9
addg(3)(0)(4)()     // 7
addg(1)(2)(4)(8)() // 15
```

```
function addg(first) {
    if (first !== undefined) {
        return function more(next) {
            if (next === undefined) {
                return first;
            }
            first += next;
            return more;
        };
    }
}

// retursion: a function returns itself
```

Write a function **liftg** that will take a binary function and apply it to many invocations.

```
liftg(mul)()        // undefined
liftg(mul)(3)()           // 3
liftg(mul)(3)(0)(4)()     // 0
liftg(mul)(1)(2)(4)(8)()  // 64
```

```
function liftg(binary) {
    return function (first) {
        if (first !== undefined) {
            return function more(next) {
                if (next === undefined) {
                    return first;
                }
                first = binary(
                    first,
                    next
                );
                return more;
            };
        }
    };
}
```

Write a function **arrayg** that will build an array from many invocations.

```
arrayg()            // []
arrayg(3)()         // [3]
arrayg(3)(4)(5)()   // [3, 4, 5]
```

```
function arrayg(first) {
    var array = [];
    function more(next) {
        if (next === undefined) {
            return array;
        }
        array.push(next);
        return more;
    }
    return more(first);
}
```

```
function arrayg(first) {
    if (first === undefined) {
        return [];
    }
    return liftg(
        function (array, value) {
            array.push(value);
            return array;
        }
    )([first]);
}
```

Make a function **continuize** that takes a unary function, and returns a function that takes a callback and an argument.

```
sqrtc = continuize(Math.sqrt);
sqrtc(alert, 81)    // 9
```

```
function continuize(unary) {
    return function (callback, arg) {
        return callback(unary(arg));
    };
}

function continuize(any) {
    return function (callback, ...x) {
        return callback(any(...x));
    };
}
```

```
function constructor(spec) {
    var that = other_constructor(spec);
    var member;
    var method = function () {
        // spec, member, method
    };
    that.method = method;
    return that;
}
```

```
function constructor(spec) {
    let {member} = spec;
    const {other} = other_constructor(spec);
    const method = function () {
        // spec, member, other, method
    };
    return Object.freeze({
        method,
        other
    });
}
```

Make an array wrapper object with methods **get**, **store**, and **append**, such that an attacker cannot get access to the private array.

```
myvector = vector();
myvector.append(7);
myvector.store(1, 5);
myvector.get(0)    // 7
myvector.get(1)    // 5
```

```
function vector() {
    var array = [];

    return {
        get: function get(i) {
            return array[i];
        },
        store: function store(i, v) {
            array[i] = v;
        },
        append: function append(v) {
            array.push(v);
        }
    };
}
```

```
function vector() {
    var array = [];

    return {
        get: function get(i) {
            return array[i];
        },
        store: function store(i, v) {
            array[i] = v;
        },
        append: function append(v) {
            array.push(v);
        }
    };    var stash;
}         myvector.store("push", function () {
              stash = this;
          });
          myvector.append(); // stash is array
```

```
function vector() {
    var array = [];

    return {
        get: function (i) {
            return array[+i];
        },
        store: function store(i, v) {
            array[+i] = v;
        },
        append: function (v) {
            array[array.length] = v;
        }
    };
}
```

Make a function that makes a publish/subscribe object. It will reliably deliver all publications to all subscribers in the right order.

```
my_pubsub = pubsub();
my_pubsub.subscribe(log);
my_pubsub.publish("It works!");
    // log("It works!")
```

```
function pubsub() {
    var subscribers = [];
    return {
        subscribe: function (subscriber) {
            subscribers.push(subscriber);
        },
        publish: function (publication) {
            var i;
            var length = subscribers.length;
            for (i = 0; i < length; i += 1) {
                subscribers[i](publication);
            }
        }
    };
}
```

```
function pubsub() {
    var subscribers = [];
    return {
        subscribe: function (subscriber) {
            subscribers.push(subscriber);
        },
        publish: function (publication) {
            var i;
            var length = subscribers.length;
            for (i = 0; i < length; i += 1) {
                subscribers[i](publication);
            }
        }
    };
}

            my_pubsub.subscribe();
```

```
function pubsub() {
    var subscribers = [];
    return {
        subscribe: function (subscriber) {
            subscribers.push(subscriber);
        },
        publish: function (publication) {
            var i;
            var length = subscribers.length;
            for (i = 0; i < length; i += 1) {
                try {
                    subscribers[i](publication);
                } catch (ignore) {}
            }
        }
    };
}
```

```
function pubsub() {
    var subscribers = [];
    return {
        subscribe: function (subscriber) {
            subscribers.push(subscriber);
        },
        publish: function (publication) {
            var i;
            var length = subscribers.length;
            for (i = 0; i < length; i += 1) {
                try {
                    subscribers[i](publication);
                } catch (ignore) {}
            }
        }
    };
}
```

```
my_pubsub.publish = undefined;
```

```
function pubsub() {
    var subscribers = [];
    return Object.freeze({
        subscribe: function (subscriber) {
            subscribers.push(subscriber);
        },
        publish: function (publication) {
            var i;
            var length = subscribers.length;
            for (i = 0; i < length; i += 1) {
                try {
                    subscribers[i](publication);
                } catch (ignore) {}
            }
        }
    });
}
```

```
function pubsub() {
    var subscribers = [];
    return Object.freeze({
        subscribe: function (subscriber) {
            subscribers.push(subscriber);
        },
        publish: function (publication) {
            var i;
            var length = subscribers.length;
            for (i = 0; i < length; i += 1) {
                try {
                    subscribers[i](publication);
                } catch (ignore) {}
            }
        }
    });     my_pubsub.subscribe(function () {
}               this.length = 0;
            });
```

```
function pubsub() {
    var subscribers = [];
    return Object.freeze({
        subscribe: function (subscriber) {
            subscribers.push(subscriber);
        },
        publish: function (publication) {
            subscribers.forEach(function (s) {
                try {
                    s(publication);
                } catch (ignore) {}
            });
        }
    });
}
```

```
function pubsub() {
    var subscribers = [];
    return Object.freeze({
        subscribe: function (subscriber) {
            subscribers.push(subscriber);
        },
        publish: function (publication) {
            subscribers.forEach(function (s) {
                try {
                    s(publication);
                } catch (ignore) {}
            });
        }
    });
}
        my_pubsub.subscribe(limit(function () {
            my_pubsub.publish("Out of order");
        }, 1));
```

```
function pubsub() {
    var subscribers = [];
    return Object.freeze({
        subscribe: function (subscriber) {
            subscribers.push(subscriber);
        },
        publish: function (publication) {
            subscribers.forEach(function (s) {
                setTimeout(function () {
                    s(publication);
                }, 0);
            });
        }
    });
}
```