Function Challenge

Table of Contents

[Part 1 – Warmup 2](#_Toc534445597)

[Exercise 1.1: Evaluate the following code, what is the output ? 2](#_Toc534445598)

[Exercise 1.2: Evaluate the following code, what is the output ? 2](#_Toc534445599)

[Part 2 - Function Challenge 2](#_Toc534445600)

[Exercise 2.1: Write an identity function that takes an argument and return that arguments. Example: 3](#_Toc534445601)

[Exercise 2.2: Write 3 binary function: add, sub, and mul, that take 2 numbers and return their sum, difference, and product. Example: 3](#_Toc534445602)

[Exercise 2.3: Write a function identifyf that takes an argument and returns a function that returns that argument. Example: 3](#_Toc534445603)

[Exercise 2.4: Write a function addf that adds from 2 invocations. Example: 3](#_Toc534445604)

[Exercise 2.5: Write a function liftf that takes a binary function, and makes it callable with 2 invocations. Example: 3](#_Toc534445605)

[Exercise 2.6: Write a function curry that takes a binary function and an argument, and returns a function that can take a second arguments. Example: 4](#_Toc534445606)

[Exercise 2.7: Without writing any new functions, show 3 ways to create the inc function that add 1 to the argument. Example: 4](#_Toc534445607)

[Exercise 2.8: Write a function twice that takes a binary function and returns a unary function that passes its argument to the binary function twice. Example: 4](#_Toc534445608)

[Exercise 2.9: Write reverse, a function that reverses the arguments of a binary function. Example: 5](#_Toc534445609)

[Exercise 2.10: Write a function composeu that takes two unary functions and returns a unary function that calls them both. Example: 5](#_Toc534445610)

[Exercise 2.11: Write a function composeb that takes two binary functions and returns a function that calls them both. Example: 5](#_Toc534445611)

[Exercise 2.12: Write a limit function that allows a binary function to be called a limited number of times. Example: 5](#_Toc534445612)

[Exercise 2.13: Write a from function that produces a generator that will produce a series of values. Example: 6](#_Toc534445613)

[Exercise 2.14: Write a to function that takes a generator and an end value, and returns another generator that will produce numbers up to that limit. Example: 6](#_Toc534445614)

[Exercise 2.15: Write a fromTo function that return a generator that produce values in a range. Example: 6](#_Toc534445615)

[Exercise 2.16: Write an element function that takes an array and a generator and returns a generator that will produce elements from the array. Example: 7](#_Toc534445616)

[Exercise 2.17: Modify the element function so that the generator argument is optional. If a generator is not provided, then each of the elements of the array will be produced. Example: 7](#_Toc534445617)

[Exercise 2.18: Write a collect function that takes a generator and array and produces a function that will collect the results in the array. Example: 8](#_Toc534445618)

[Exercise 2.19: Write a filter function that takes a generator and a predicate and produces a generator that produces only the values approved by the predicate. Example: 8](#_Toc534445619)

[Exercise 2.20: Write a concat function that takes 2 generator and produces a generator that combines the sequences. Example: 8](#_Toc534445620)

[Exercise 2.21: Make a function gensymf that makes a function that generatres unique symbols. Example: 9](#_Toc534445621)

[Exercise 2.22: Write a function gensymff that takes a unary function and a seed and returns a gensymf. Example: 9](#_Toc534445622)

[Exercise 2.23: Make a function fibonaccif that returns a generator that will return the next fibonacci number. Example: 10](#_Toc534445623)

[Exercise 2.24: Write a counter function that returns an object containing 2 functions that implement an up/down counter, hiding the counter. Example: 10](#_Toc534445624)

[Exercise 2.25: Make a revocable function 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. Example: 11](#_Toc534445625)

[Exercise 2.26: Write a function exp that evaluates simple array expressions. Example: 11](#_Toc534445626)

[Exercise 2.27: Modify exp to evaluate nested array expressions. Example: 12](#_Toc534445627)

[Exercise 2.28: Write a function addg that adds from many invocations, until it sees an empty invocation. Example: 12](#_Toc534445628)

[Exercise 2.29: Write a function liftg that takes a binary function and apply it to many invocations. Example: 13](#_Toc534445629)

[Exercise 2.30: Write a function arrayg that will build an arrayg from many invocations. Example: 13](#_Toc534445630)

# 

# **Part 1 – Warmup**

# **Exercise 1.1**: Evaluate the following code, what is the output ?

**function** **empty**(o) {

o = null;

}

**var** x = [];

empty(x);

console.log(x); // ??

# **Exercise 1.2**: Evaluate the following code, what is the output ?

**function** **swap**(a, b) {

[a, b] = [b, a];

}

**var** x = 1;

**var** y = 2;

swap(x, y);

console.log(x); // ??

# **Part 2 - Function Challenge**

# **Exercise 2.1**: Write an **identity** function that takes an argument and return that arguments. Example:

identify(3); // 3

# **Exercise 2.2**: Write 3 binary function: **add**, **sub**, and **mul**, that take 2 numbers and return their sum, difference, and product. Example:

add(3, 4); // 3 + 4 = 7

sub(3, 4); // 3 – 4 = -1

mul(3, 4); // 3 \* 4 = 12

# **Exercise 2.3**: Write a function **identifyf** that takes an argument and returns a function that returns that argument. Example:

**var** four = identifyf(4);

four(); // 4

# 

# **Exercise 2.4**: Write a function **addf** that adds from 2 invocations. Example:

addf(3)(4); // 7

# **Exercise 2.5**: Write a function **liftf** that takes a binary function, and makes it callable with 2 invocations. Example:

**var** addf = liftf(add);

addf(3)(4); // 7

liftf(mul)(5)(6); // 30

# **Exercise 2.6**: Write a function **curry** that takes a binary function and an argument, and returns a function that can take a second arguments. Example:

**var** add3 = curry(add, 3);

add3(4); // 7

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

# **Exercise 2.7**: Without writing any new functions, show 3 ways to create the **inc** function that add 1 to the argument. Example:

**var** inc = \_\_\_\_ // TODO;

inc(5); // 6

inc(inc(5)); // 7

# **Exercise 2.8**: Write a function **twice** that takes a binary function and returns a unary function that passes its argument to the binary function twice. Example:

add(11, 11); // 22

**var** doubl = twice(add);

doubl(11); // 22

**var** square = twice(mul);

square(11); // 121

# **Exercise 2.9**: Write **reverse**, a function that reverses the arguments of a binary function. Example:

**var** bus = reverse(sub);

bus(3, 2) // -1

# **Exercise 2.10**: Write a function **composeu** that takes two unary functions and returns a unary function that calls them both. Example:

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

// doubl(5) => 5 + 5 = 10

// square(10) => 100

# **Exercise 2.11**: Write a function **composeb** that takes two binary functions and returns a function that calls them both. Example:

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

// add(2, 3) => 5

// mul(5, 7) => 35

# **Exercise 2.12**: Write a **limit** function that allows a binary function to be called a limited number of times. Example:

**var** addLimited = limit(add, 1);

addLimited(3, 4); // 7

addLimited(3, 5); // undefined

addLimited(4, 5); // undefined

# 

# **Exercise 2.13**: Write a **from** function that produces a generator that will produce a series of values. Example:

**var** index = from(0);

index(); // 0

index(); // 1

index(); // 2

# 

# **Exercise 2.14**: Write a **to** function that takes a generator and an end value, and returns another generator that will produce numbers up to that limit. Example:

**var** index = to(from(1), 3);

index(); // 1

index(); // 2

index(); // undefined

index(); // undefined

# 

# **Exercise 2.15**: Write a **fromTo** function that return a generator that produce values in a range. Example:

**var** index = fromTo(0, 3);

index(); // 0

index(); // 1

index(); // 2

index(); // undefined

# **Exercise 2.16**: Write an **element** function that takes an array and a generator and returns a generator that will produce elements from the array. Example:

**var** ele = element([1, 2, 3, 4], fromTo(1, 3));

ele(); // 2 => index 1

ele(); // 3 => index 2

ele(); // undefined => index 3

# **Exercise 2.17:** Modify the **element** function so that the generator argument is optional. If a generator is not provided, then each of the elements of the array will be produced. Example:

**var** ele = element([1, 2, 3, 4]);

ele(); // 1

ele(); // 2

ele(); // 3

ele(); // 3

ele(); // undefined

# **Exercise 2.18**: Write a **collect** function that takes a generator and array and produces a function that will collect the results in the array. Example:

**var** array = [];

**var** col = collect(fromTo(0, 2), array);

col(); // 0

col(); // 1

col(); // undefined

array; // [0, 1]

# **Exercise 2.19**: Write a **filter** function that takes a generator and a predicate function and produces a generator that produces only the values approved by the predicate. Example:

**var** fil = filter(fromTo(0, 5),

**function** **even**(value) {

**return** (value % 2) === 0;

});

fil(); // 0

fil(); // 2

fil(); // 4

fil(); // undefined

# **Exercise 2.20**: Write a **concat** function that takes 2 generator and produces a generator that combines the sequences. Example:

**var** con = concat(fromTo(0, 3), fromTo(0, 2));

con(); // 0

con(); // 1

con(); // 2

con(); // 0

con(); // 1

con(); // undefined

# **Exercise 2.21**: Make a function **gensymf** that makes a function that generatres unique symbols. Example:

**var** genG = gensymf('G');

**var** genH = gensymf('H');

genG(); // G1

genG(); // G2

genH(); // H1

genH(); // H2

# **Exercise 2.22**: Write a function **gensymff** that takes a unary function and a seed and returns a gensymf. Example:

**var** gensymf = gensymff(inc, 0);

**var** genG = gensymf('G');

**var** genH = gensymf('H');

genG(); // G1

genG(); // G2

genH(); // H1

genH(); // H2

# 

# **Exercise 2.23**: Make a function fibonaccif that returns a generator that will return the next fibonacci number. Example:

**var** fib = fibonaccif(0, 1); // must by fibonaccif(0, 1)

fib(); // 0

fib(); // 1

fib(); // 1

fib(); // 2

fib(); // 3

fib(); // 5

# **Exercise 2.24**: Write a **counter** function that returns an object containing 2 functions that implement an up/down counter, hiding the counter. Example:

**var** obj = counter(10);

**var** up = obj.up;

**var** down = obj.down();

up(); // 11

down(); // 10

down(); // 9

up(); // 10

# 

# **Exercise 2.25**: Make a **revocable** function 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. Example:

**var** rev = revocable(add);

**var** addRev = rev.invoke;

addRev(3, 4); // 7

rev.revoke();

addRev(3, 4); // undefined

# **Exercise 2.26**: Write a function **exp** that evaluates simple array expressions. Example:

**var** sae = [mul, 5, 11];

exp(sae); // 55

exp(42); // 42

# 

# **Exercise 2.27**: Modify **exp** to evaluate nested array expressions. Example:

**var** nae = [

Math.sqrt,

[ add,

[square, 3],

[square, 4]

]

];

exp(nae); // 5

# **Exercise 2.28**: Write a function **addg** that adds from many invocations, until it sees an empty invocation. Example:

addg(); // undefined

addg(2)(); // 2

addg(2)(7)(); // 9

addg(3)(0)(4)(); // 7

addg(2)(); // 2

# **Exercise 2.29:** Write a function **liftg** that takes a binary function and apply it to many invocations. Example:

liftg(mul)(); // undefined

liftg(mul)(3)(); // 3

liftg(mul)(3)(0)(4)(); // 0

liftg(mul)(1)(2)(4)(8)(); // 64

# 

# **Exercise 2.30**: Write a function **arrayg** that will build an array from many invocations. Example:

arrayg(); // []

arrayg(3)(4)(); // [3, 4]

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