PSCS

Problem Solving for Computer Science

IS51021C

Goldsmiths Computing

March 15, 2021

Wk 9

Problem 6:

You've been given the task of helping to build a "pre-compiler" for a JavaScript teaching tool

This will conduct preliminary checks on code to look for syntax errors to avoid compile errors

Your part in this project is to write code that checks for bracketing errors, both {} and ()

Describe an algorithm and/or JavaScript implementation that flags an error when there is a bracketing error in the code

Problem 6:

Your part in this project is to write code that checks for bracketing errors, both {} and ()

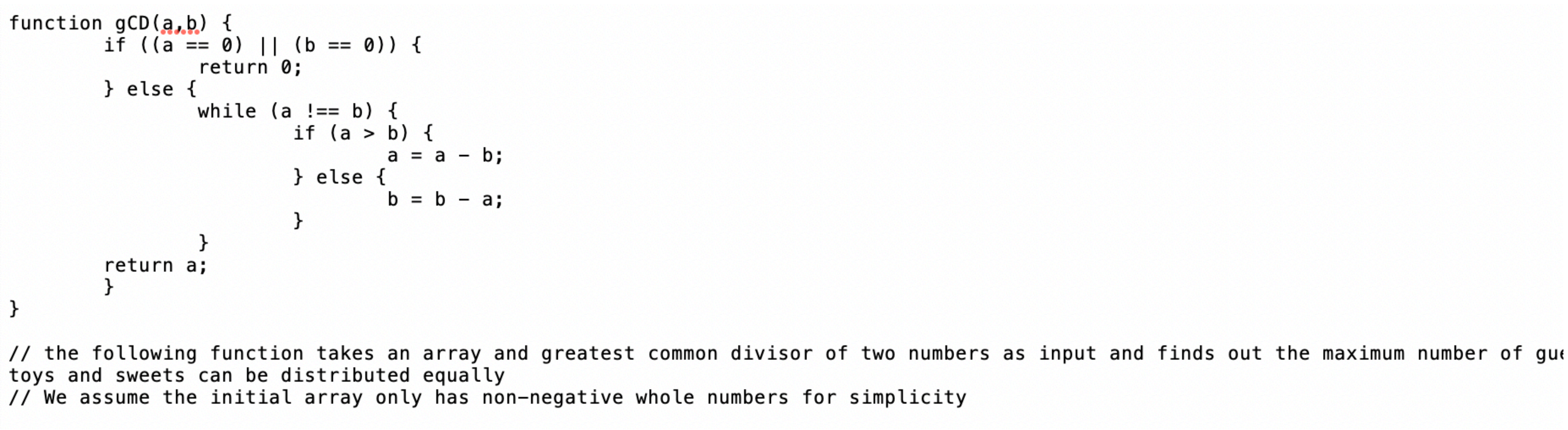
Describe an algorithm and/or JavaScript implementation that flags an error when there is a bracketing error in the code

Assume code can written into a string…

Problem 6:

Your part in this project is to write code that checks for bracketing errors, both {} and ()

Describe an algorithm and/or JavaScript implementation that flags an error when there is a bracketing error in the code



```
function gCD(a,b) {
        if ((a == 0) || (b == 0)) {
                return 0;
        } else {
                while (a !== b) {
                        if (a > b) {
                                a = a - b;
                        } else {
                                b = b - a;
                        }
                }
        return a;
        }
}

// the following function takes an array and greatest common divisor of two numbers as input and finds out the maximum number of gu
toys and sweets can be distributed equally
// We assume the initial array only has non-negative whole numbers for simplicity
```

…{…((…)(…)){…}…{…(…){…(…){…}…{…}}…}}…

Problem 6:

…{…((…)(…)){…}…{…(…){…(…){…}…{…}}…}}…

) ‘cancels out’ (

} ‘cancels out’ {

Problem 6:

…{…((…)(…)){…}…{…(…){…(…){…}…{…}}…}}…

) ‘cancels out’ (} ‘cancels out’ {

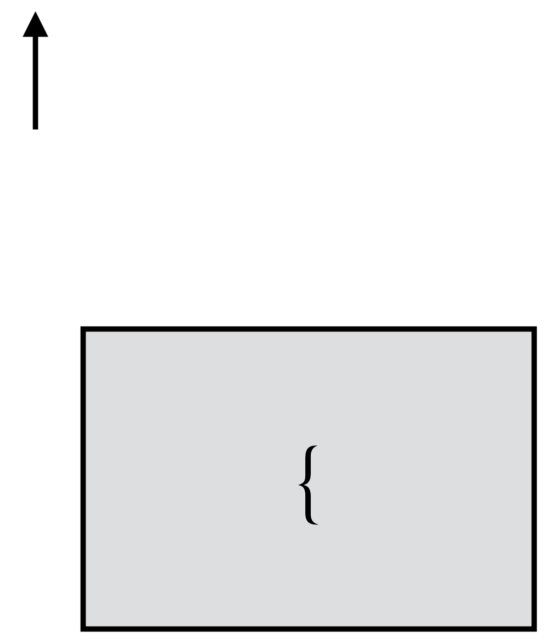
↓

…{…(…)…{…{…}…}}…

Only need to check the previous bracket
Use a stack!

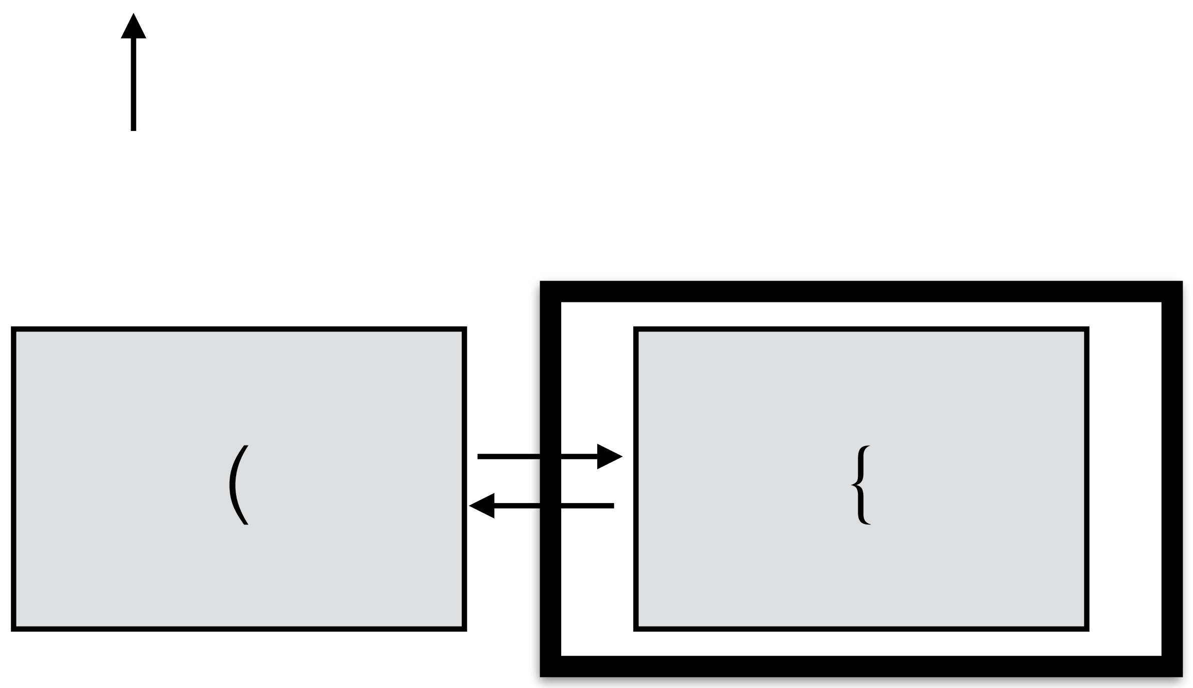
Problem 5:

…{…((…)(…)){…}…{…(…){…(…){…}…{…}}…}}…



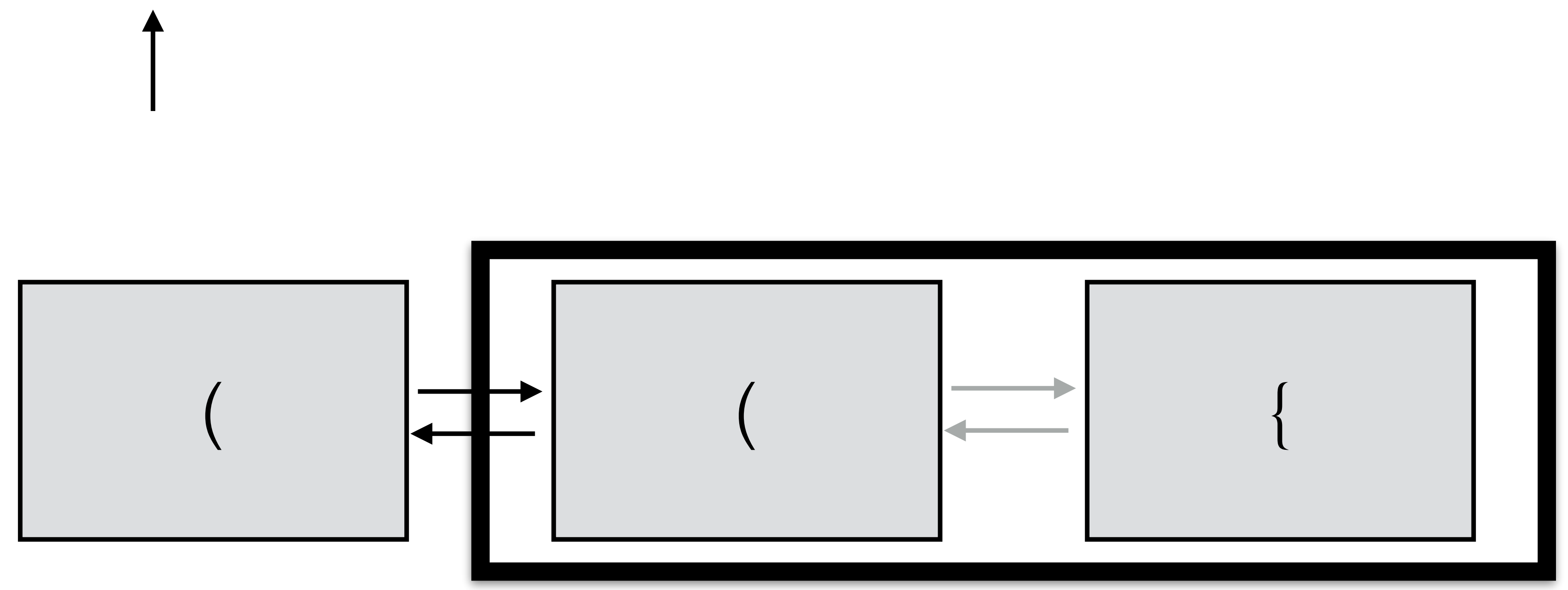
Problem 5:

...{...((...)(...)){...}...{...(...){...(...){...}...{...}}...}}...



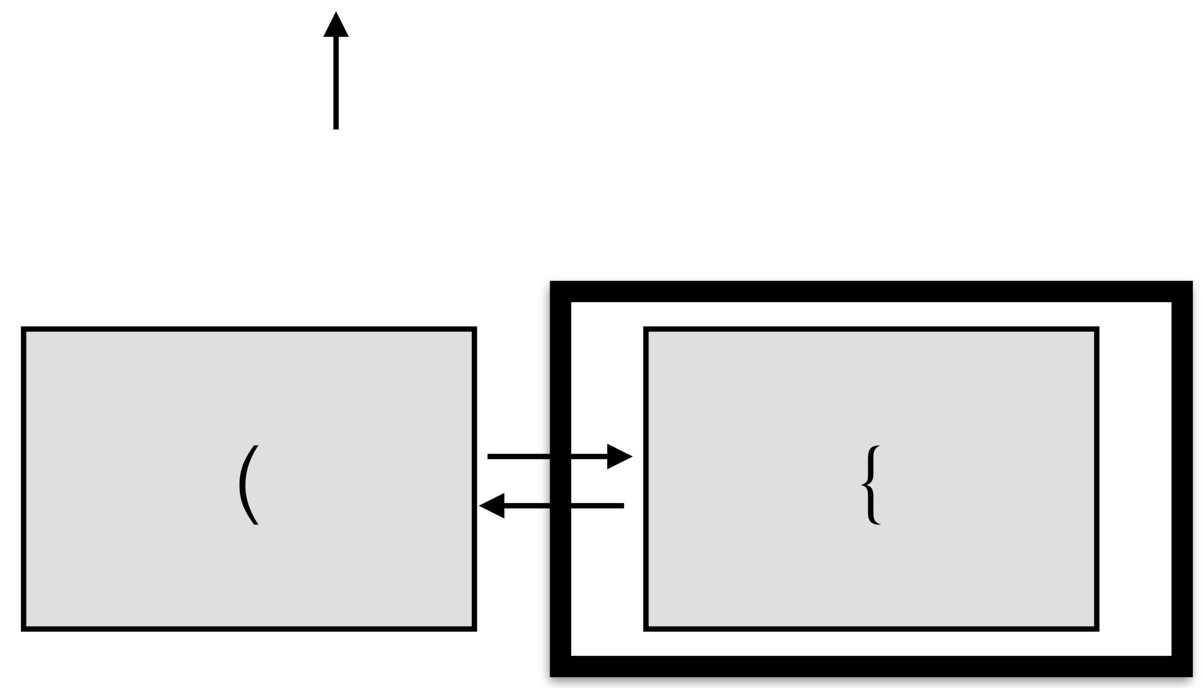
Problem 5:

...{...((...)(...)){...}...{...(...){...(...){...}...{...}}...}}...



Problem 5:

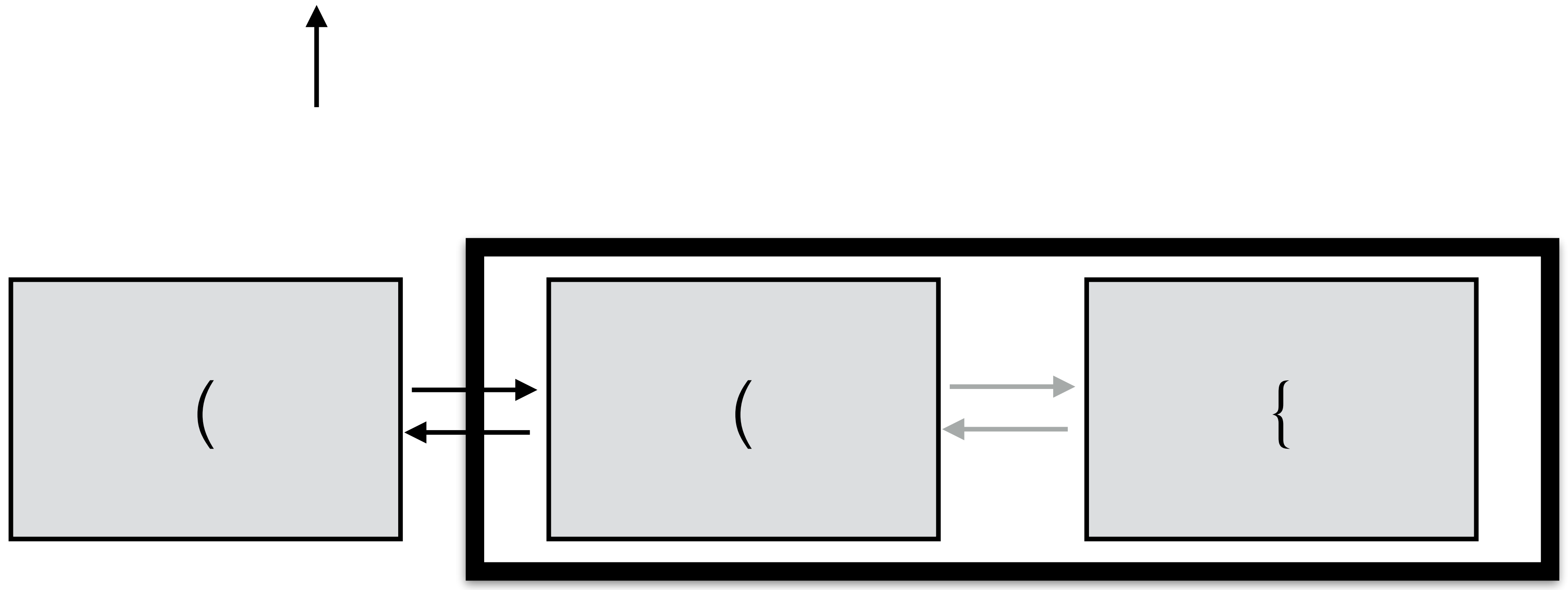
…{…((…)(…)){…}…{…(…){…(…){…}…{…}}…}}…



Pop the stack

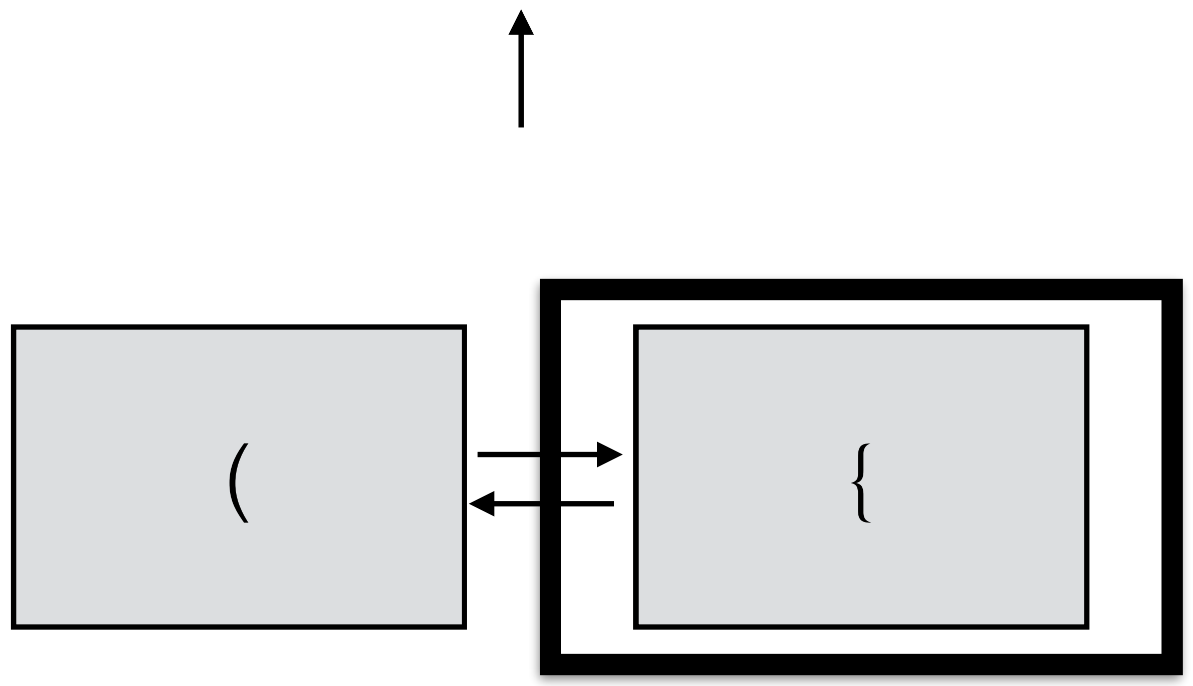
Problem 5:

…{…((…)(…)){…}…{…(…){…(…){…}…{…}}…}}…



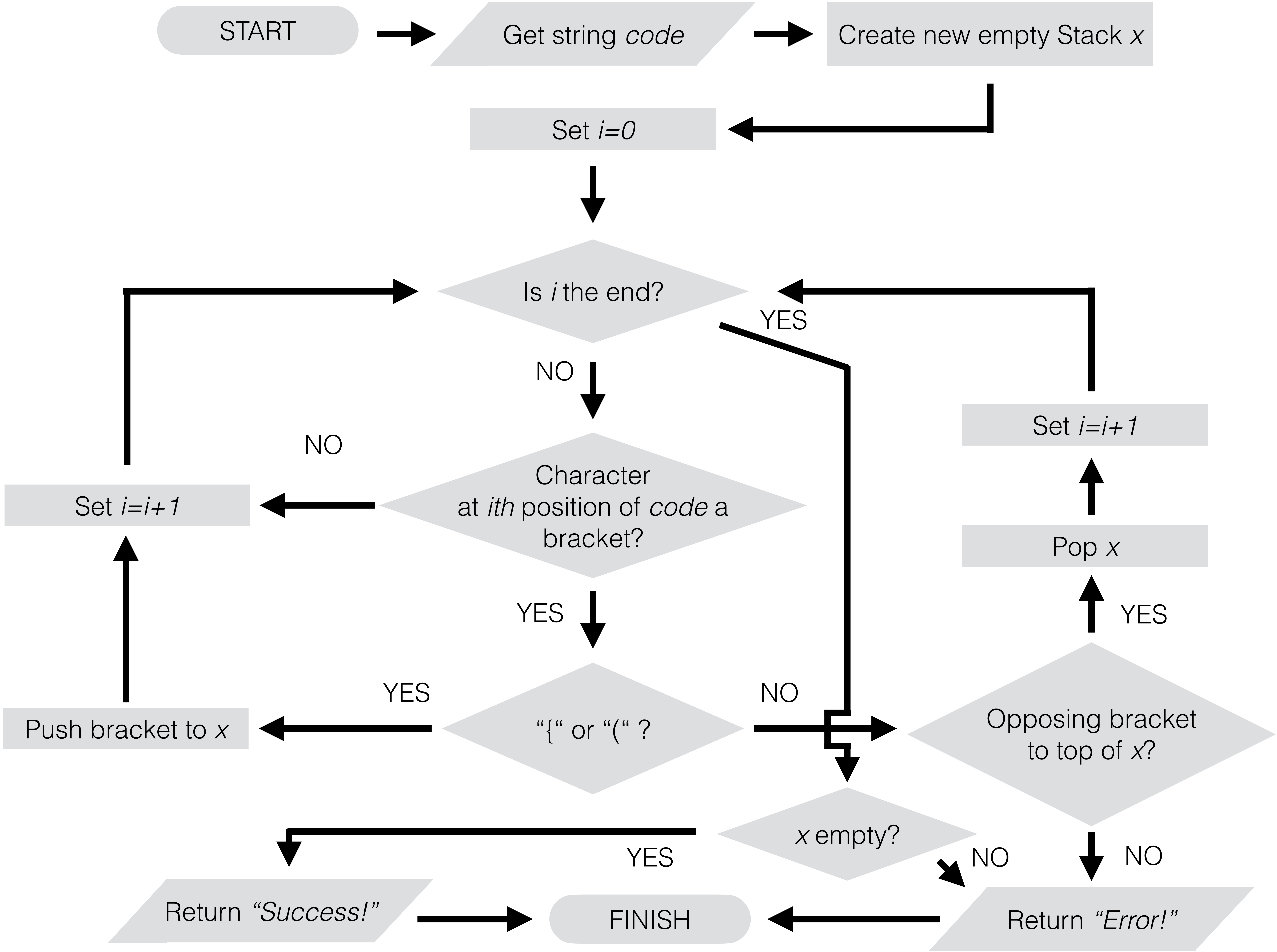
Problem 5:

…{…((…)(…)){…}…{…(…){…(…){…}…{…}}…}}…



Pop the stack

… and so on



START

Get string *code*

Create new empty Stack *x*

Set *i=0*

Is *i* the end?

NO

YES

Character at *ith* position of *code* a bracket?

NO

Set *i=i+1*

YES

“{“ or “(“ ?

YES

Push bracket to *x*

NO

Opposing bracket to top of *x*?

YES

Pop *x*

Set *i=i+1*

NO

x empty?

YES

NO

Return *“Success!”*

FINISH

Return *“Error!”*

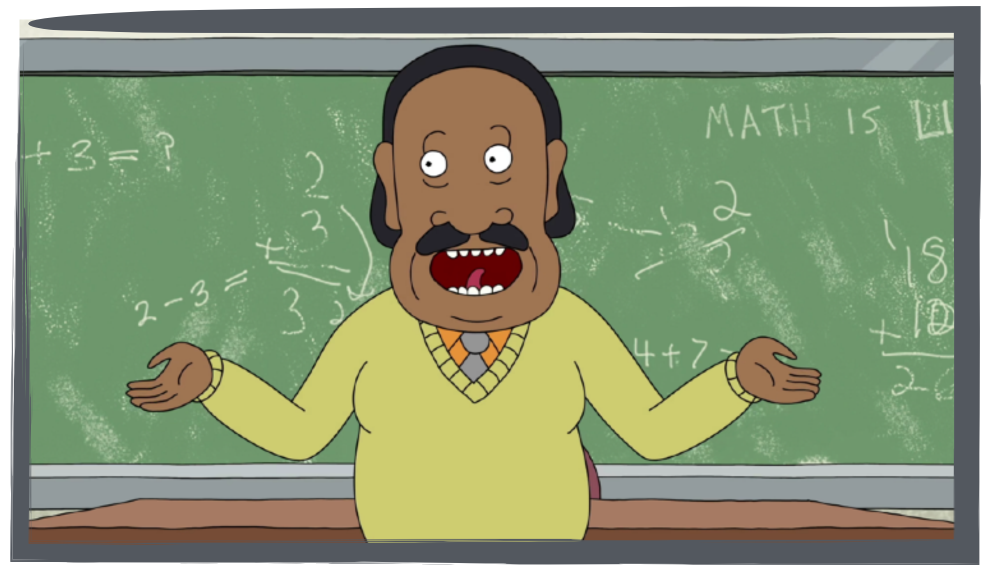
Stack demo

This problem but in the context of html tags:

http://igor.doc.gold.ac.uk/~afior002/balanced_stack/index.html

The module so far

THEORY



EXPERIMENT

Flowcharts

Searching

Algorithms

Recursion

Sorting

Functions

Node.js

Programming in JavaScript

Queues

Stacks

Abstract Data Structures

Vectors & Dynamic Arrays

Arrays & Objects

Command line

Computer model

Analysis

Time complexity

Recursion

Recursion

Deep idea in language, mathematics and computer science

Self-referential: a thing is defined in terms of itself

For a definition of recursion watch this lecture on recursion

Recursion is difficult to understand at first

- Everything you can do with loops you can do with recursion, and vice versa

So why care?

Recursion is difficult to understand at first

- Everything you can do with loops you can do with recursion, and vice versa

So why care?

- Can simplify code, e.g. loops replaced
- Gives a method of reducing problems to smaller ones
- Very useful for algorithms
- Some languages are solely recursion-based
- Recursion alone is enough to be “Turing complete”

General Form

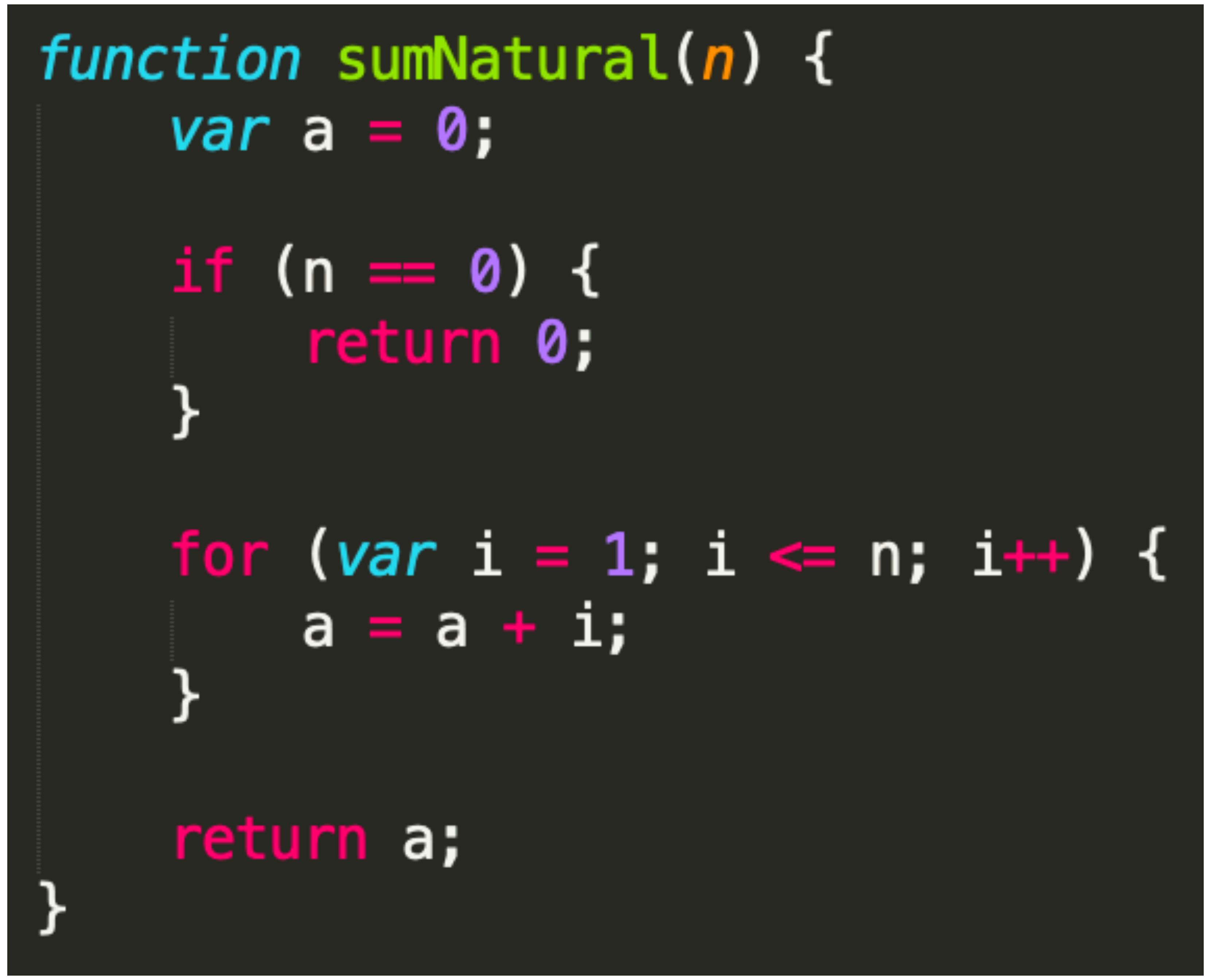
Write a function that calls the same function **within itself**

That call is made on **different** input parameters

There is a **base case** to prevent infinite recursion

Sum of all natural numbers up to and including n

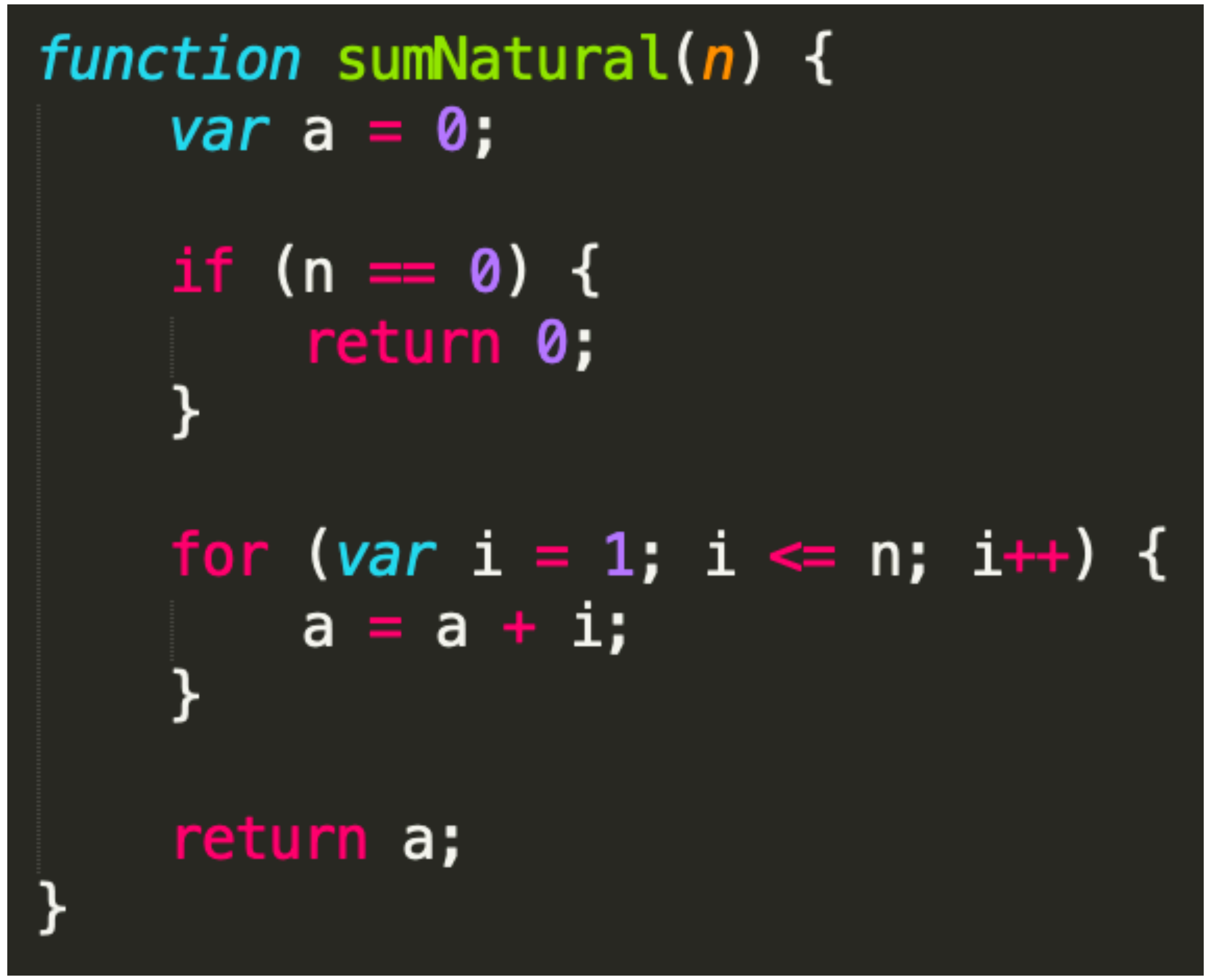
Iterative



```
function sumNatural(n) {
    var a = 0;

    if (n == 0) {
        return 0;
    }

    for (var i = 1; i <= n; i++) {
        a = a + i;
    }

    return a;
}
```

Sum of all natural numbers up to and including n

Iterative

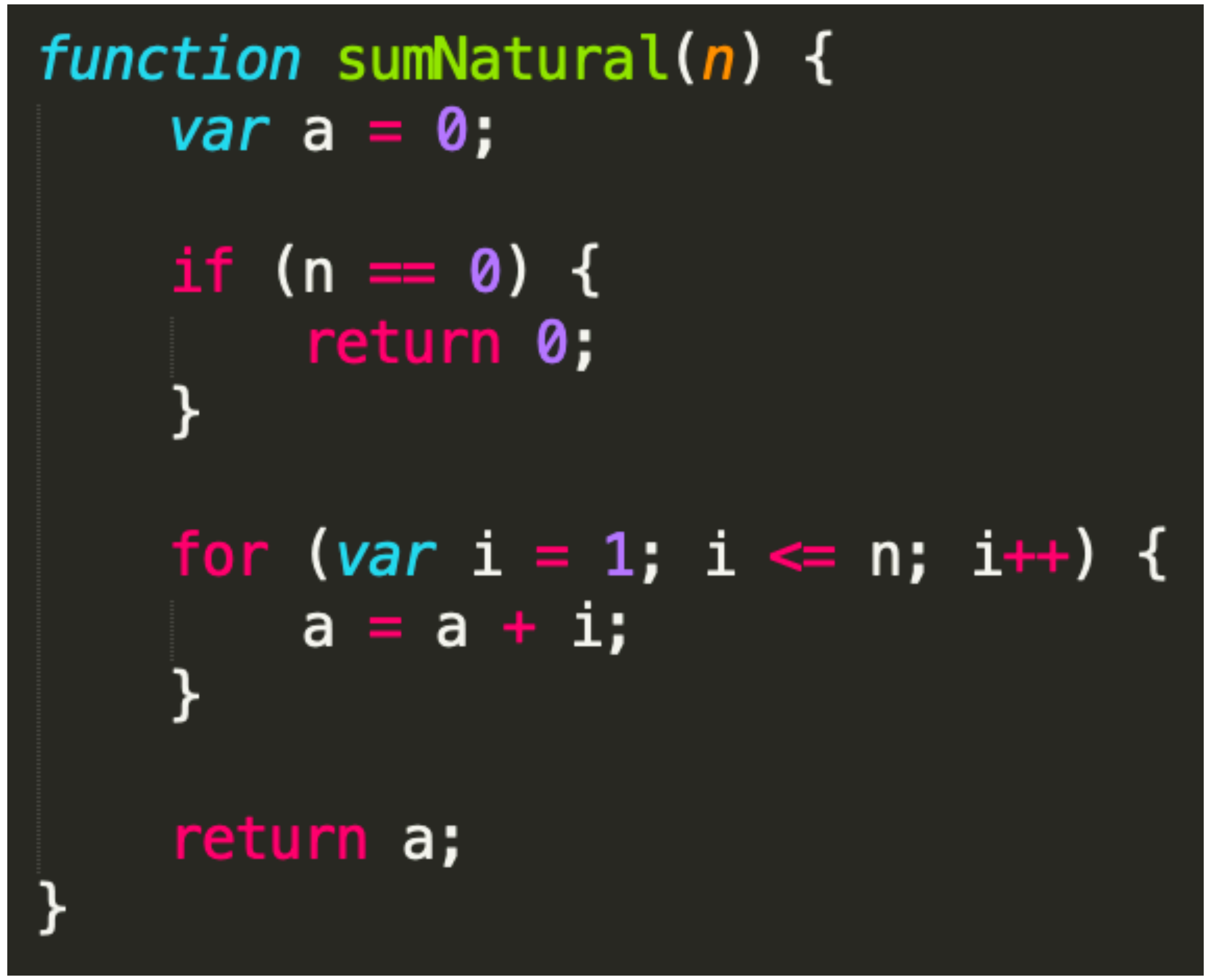


```
function sumNatural(n) {
    var a = 0;

    if (n == 0) {
        return 0;
    }

    for (var i = 1; i <= n; i++) {
        a = a + i;
    }

    return a;
}
```

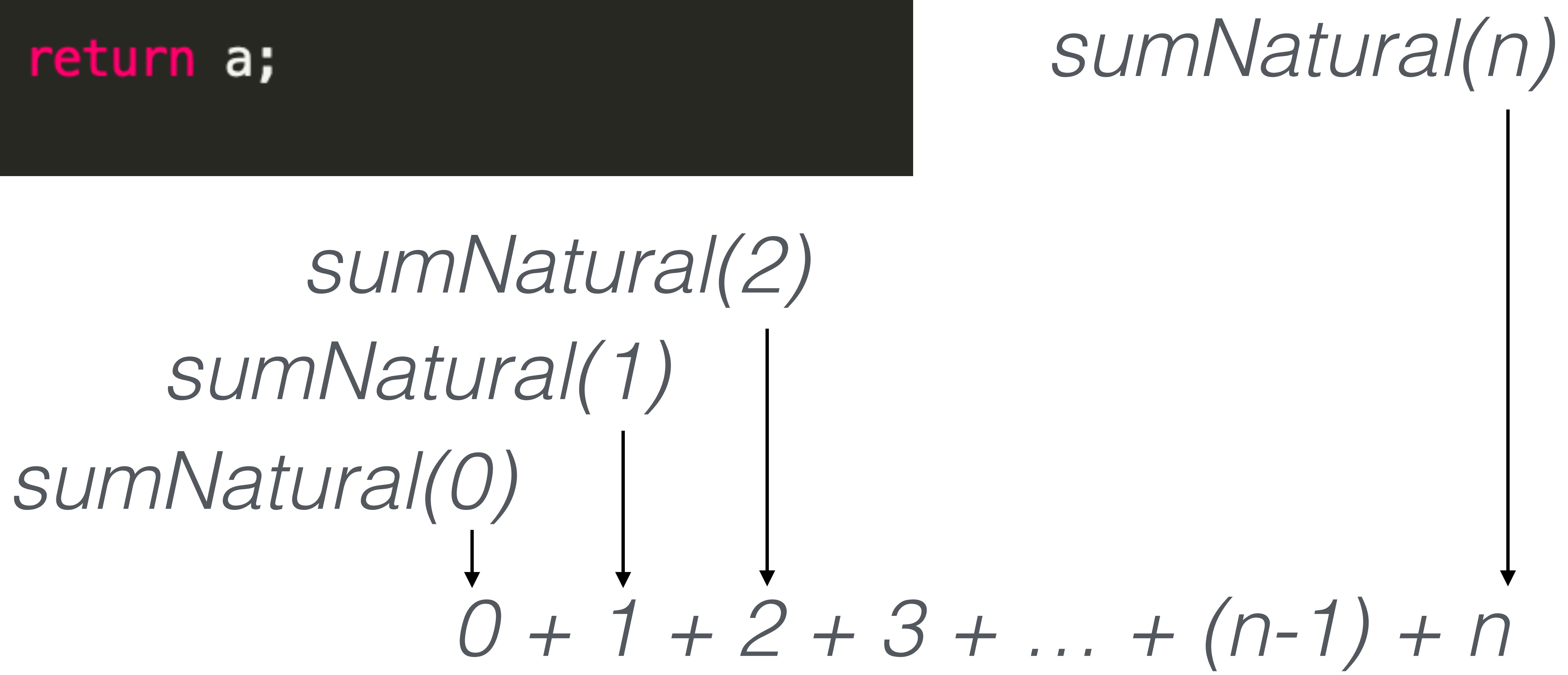
$0 + 1 + 2 + 3 + \ldots + (n-1) + n$

Sum of all natural numbers up to and including n

Iterative



```
function sumNatural(n) {
    var a = 0;

    if (n == 0) {
        return 0;
    }

    for (var i = 1; i <= n; i++) {
        a = a + i;
    }

    return a;
}
```



sumNatural(n)

sumNatural(2)

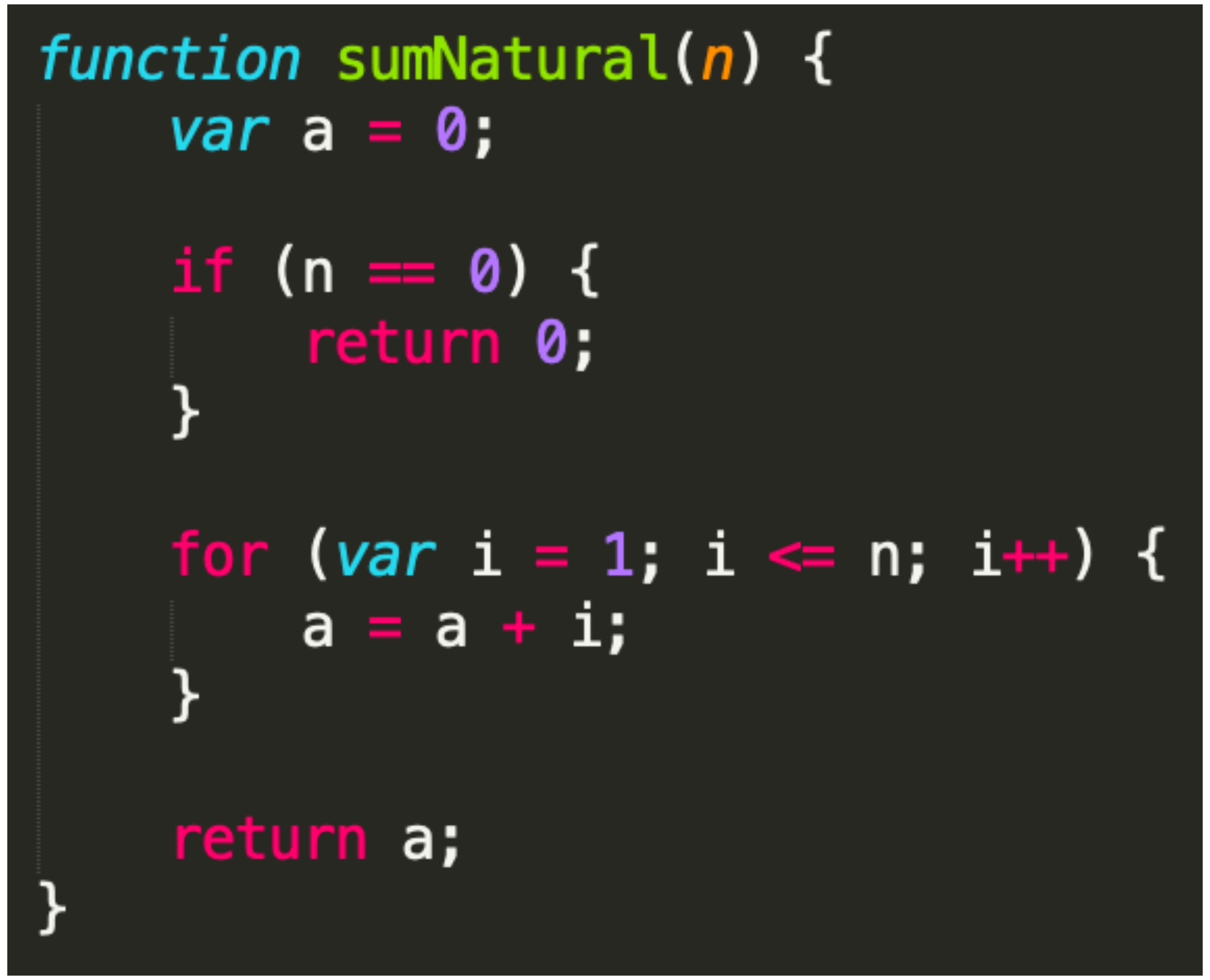
sumNatural(1)

sumNatural(0)

0 + 1 + 2 + 3 + … + (n-1) + n

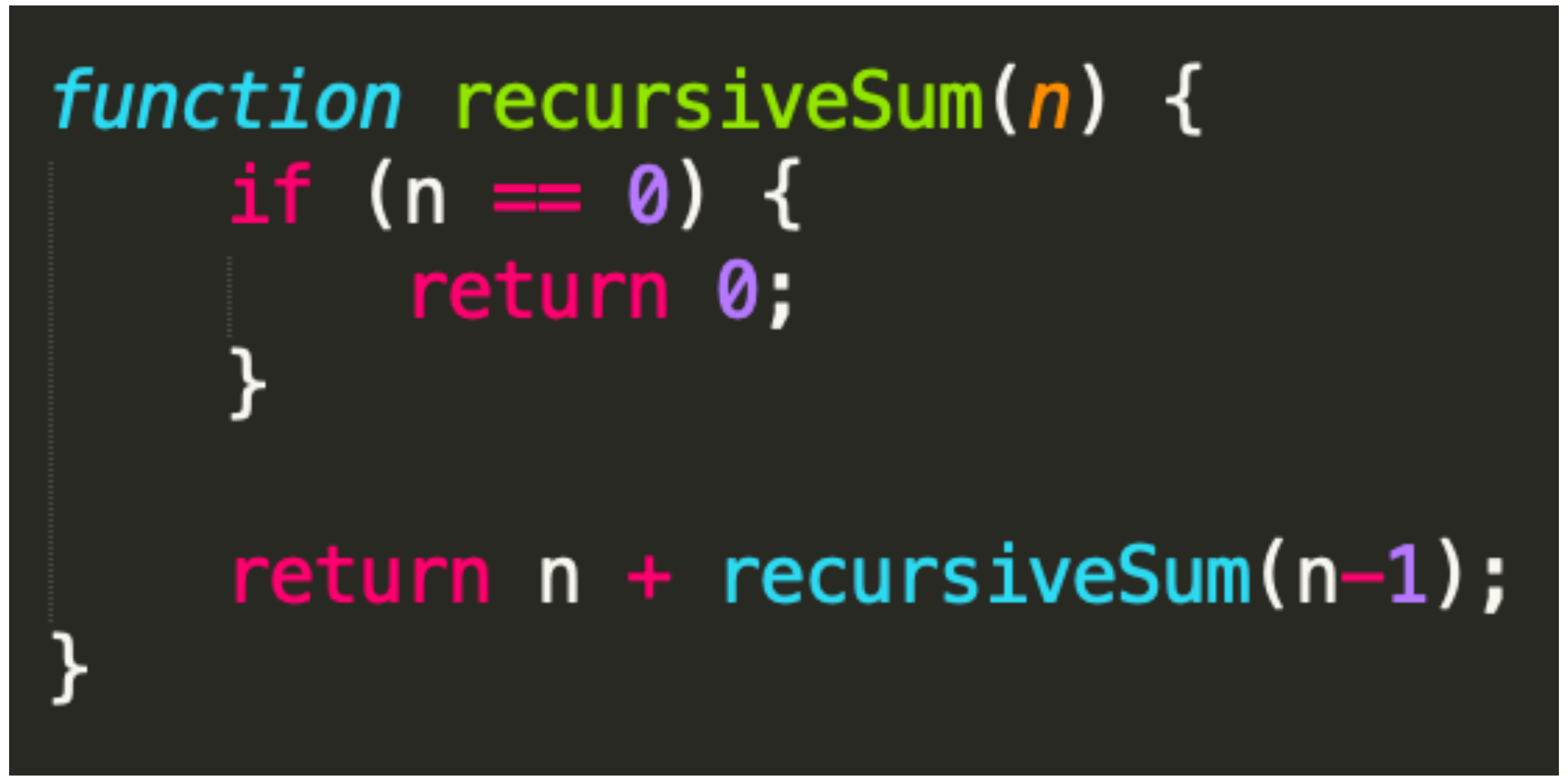
Sum of all natural numbers up to and including n

Iterative



```
function sumNatural(n) {
    var a = 0;

    if (n == 0) {
        return 0;
    }

    for (var i = 1; i <= n; i++) {
        a = a + i;
    }

    return a;
}
```

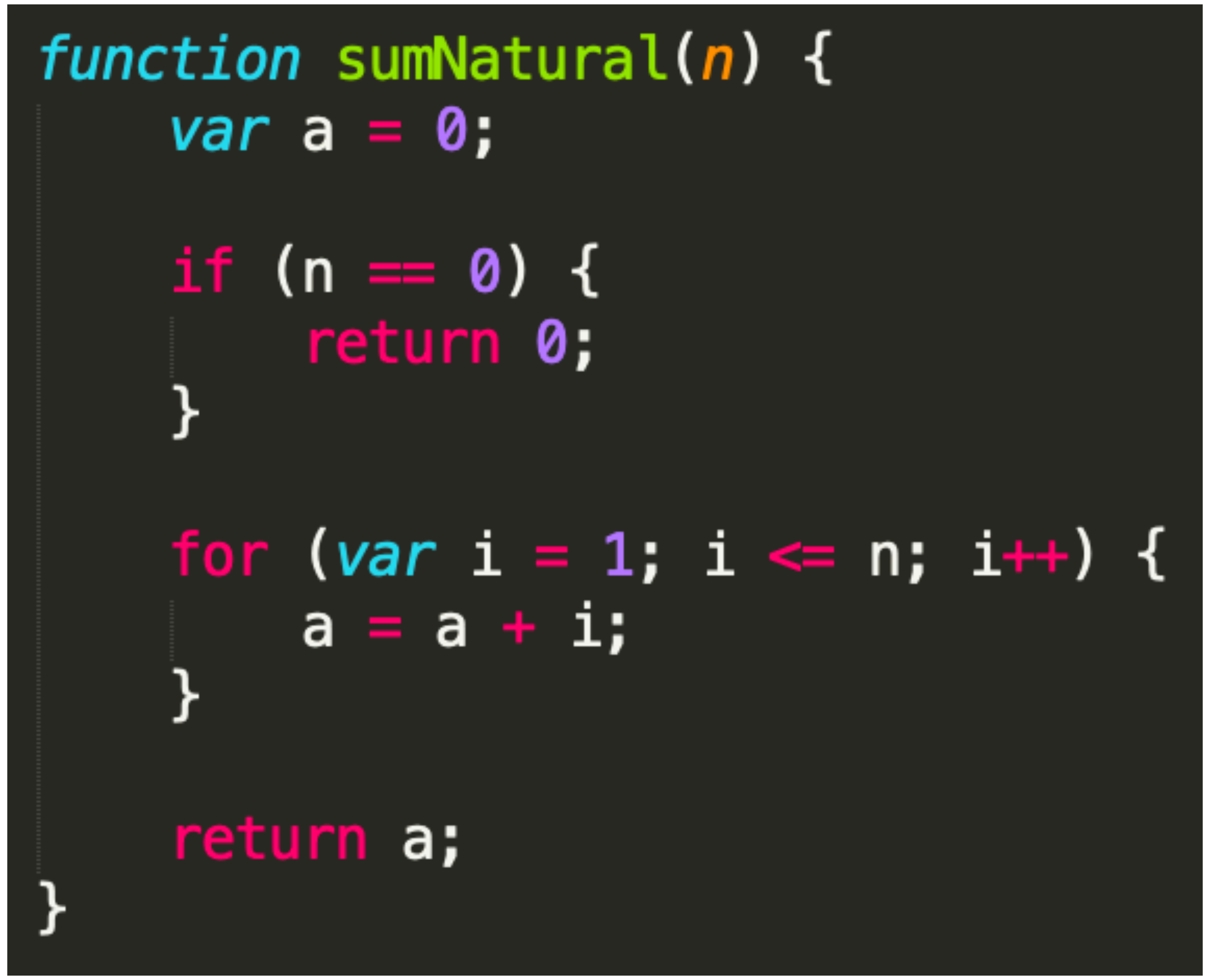
Recursive



```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n-1);
}
```

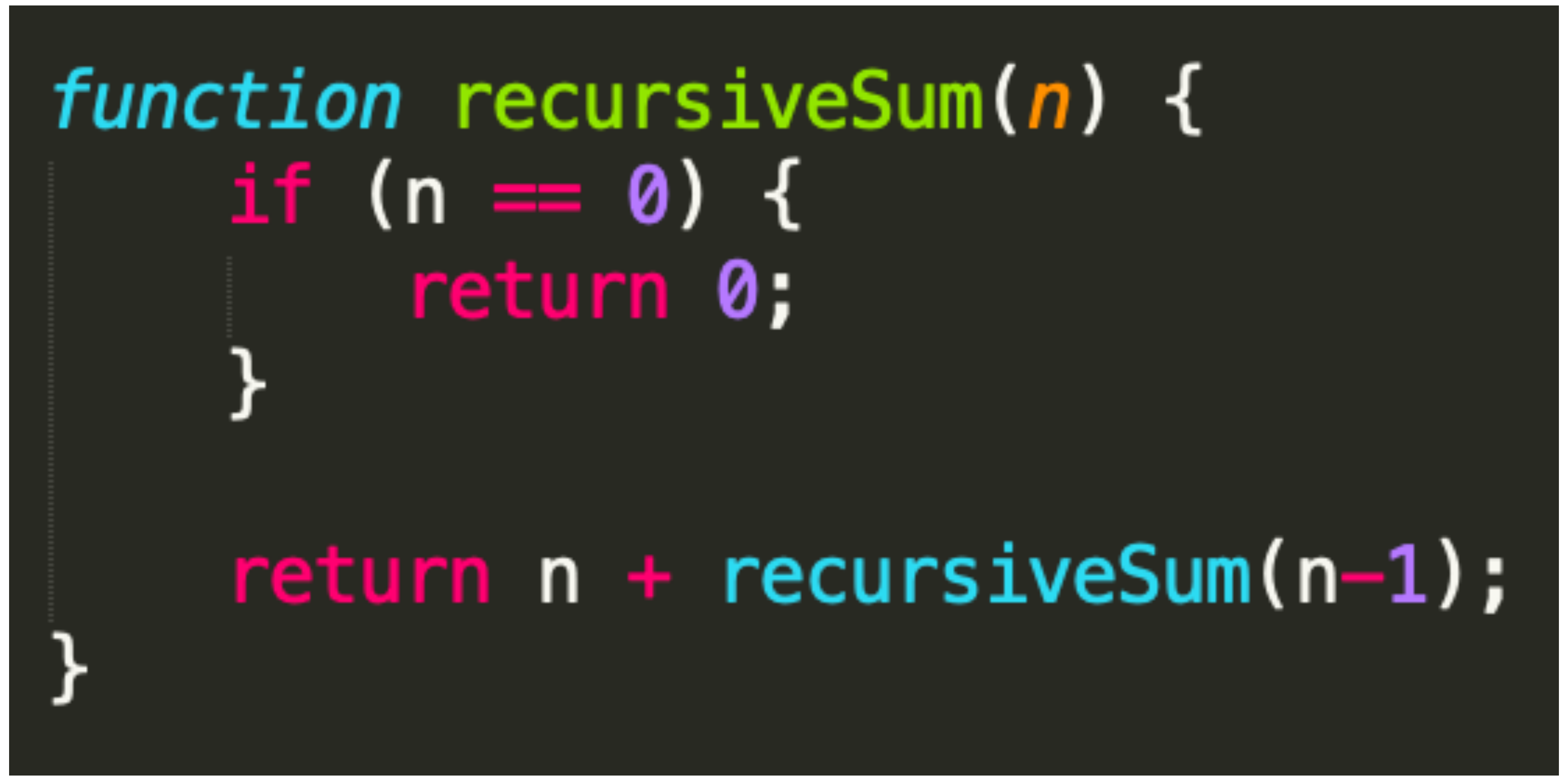
Sum of all natural numbers up to and including n

Iterative

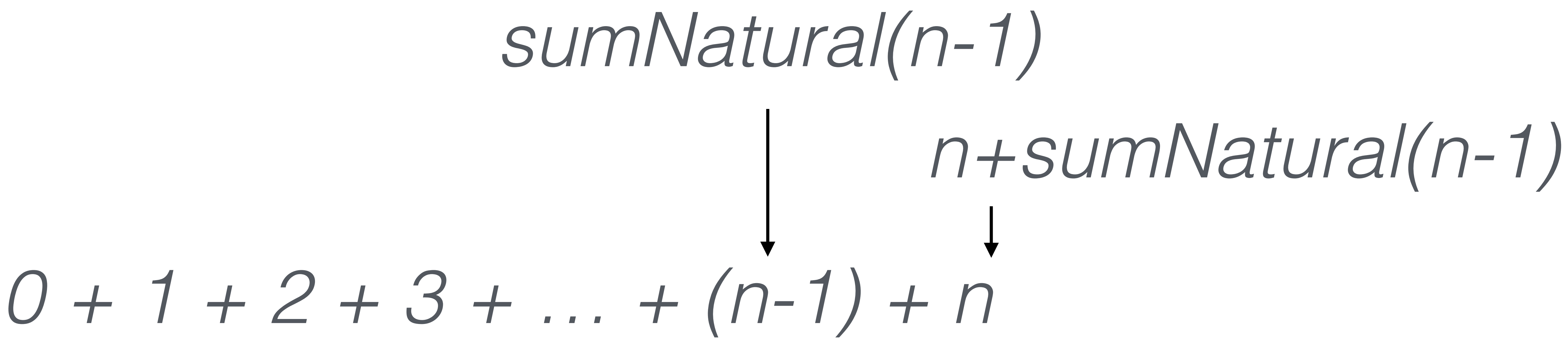


```
function sumNatural(n) {
    var a = 0;

    if (n == 0) {
        return 0;
    }

    for (var i = 1; i <= n; i++) {
        a = a + i;
    }

    return a;
}
```

Recursive



```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n-1);
}
```



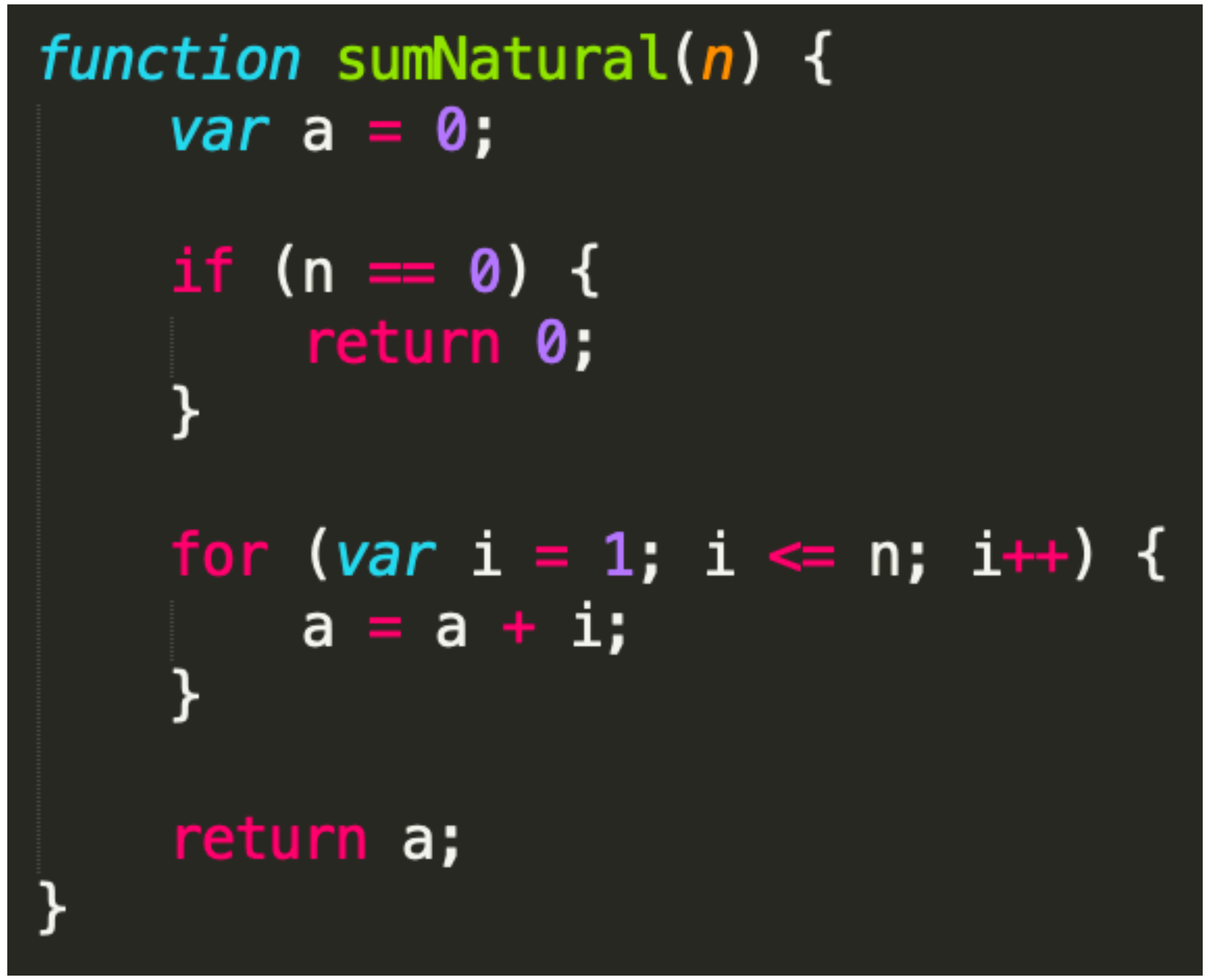
sumNatural(n-1)

n+sumNatural(n-1)

0 + 1 + 2 + 3 + … + (n-1) + n

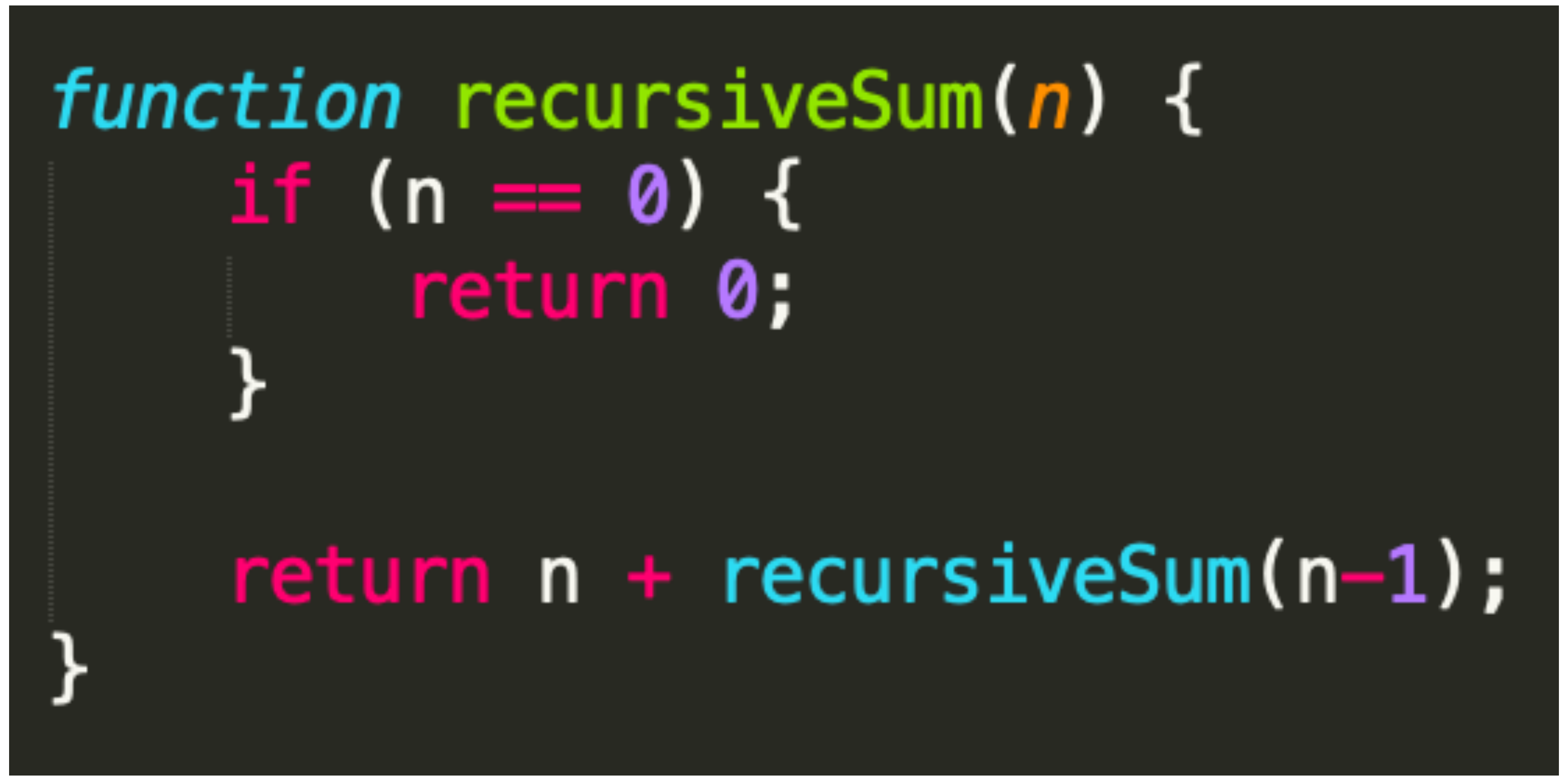
Sum of all natural numbers up to and including n

Iterative

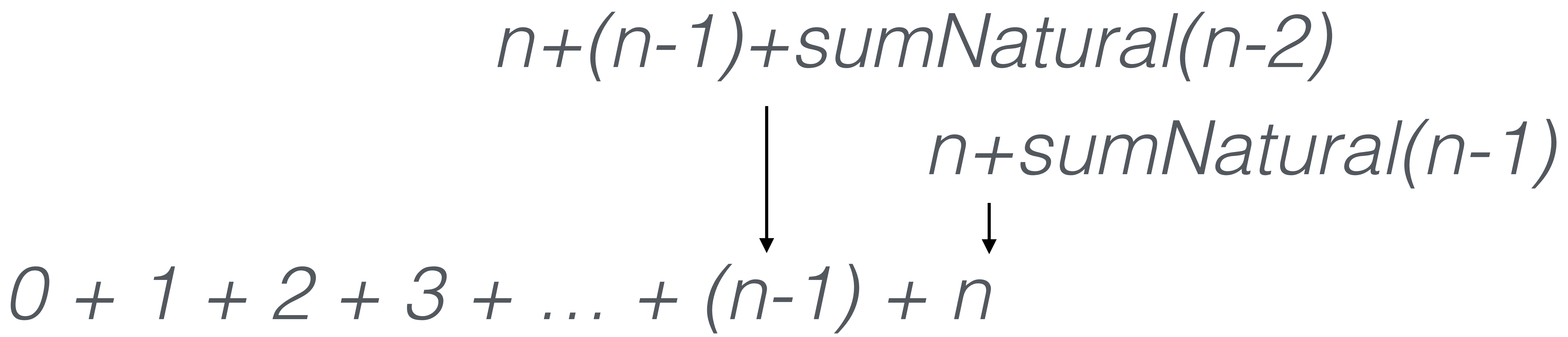


```
function sumNatural(n) {
    var a = 0;

    if (n == 0) {
        return 0;
    }

    for (var i = 1; i <= n; i++) {
        a = a + i;
    }

    return a;
}
```

Recursive



```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n-1);
}
```

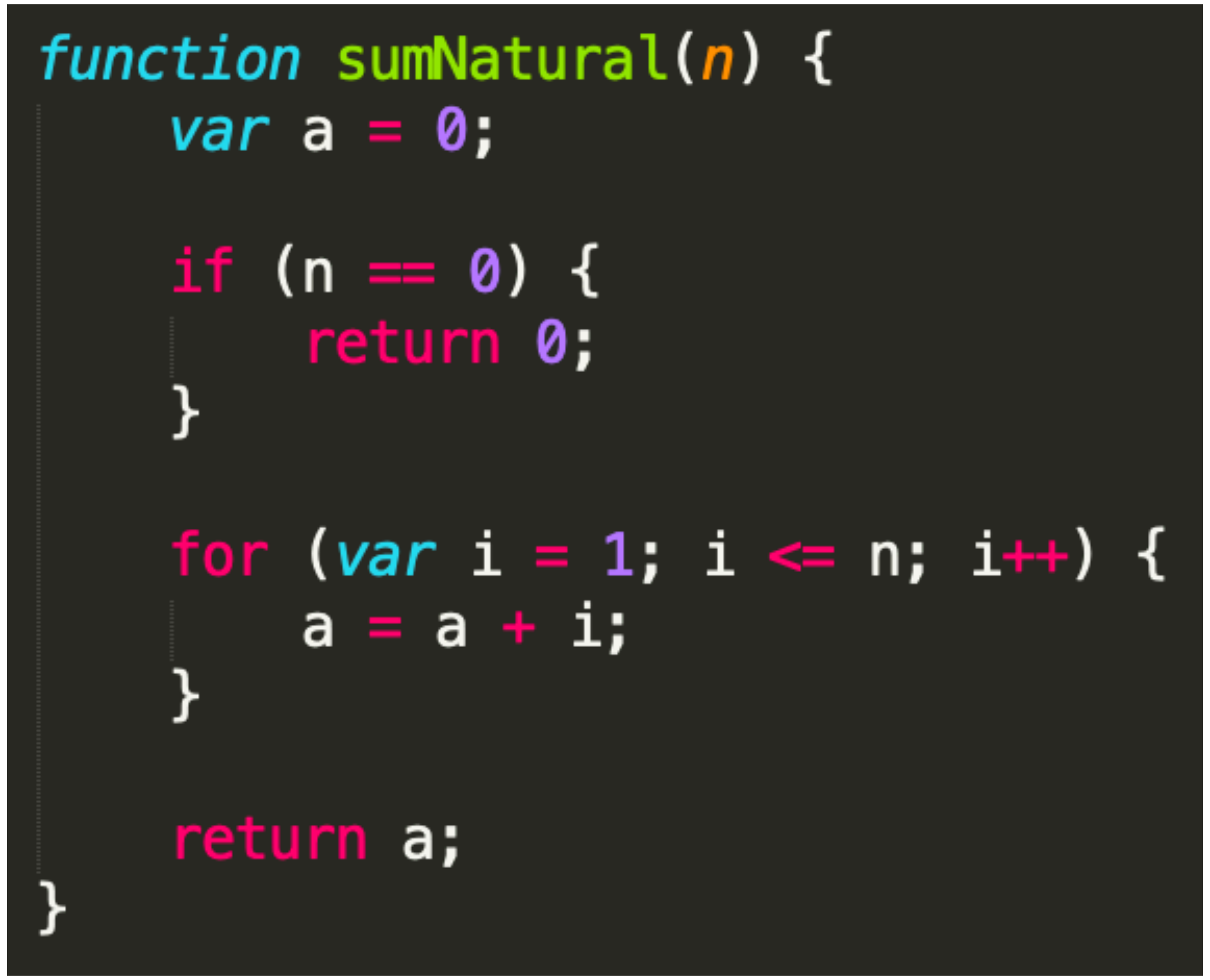


n+(n-1)+sumNatural(n-2)

n+sumNatural(n-1)

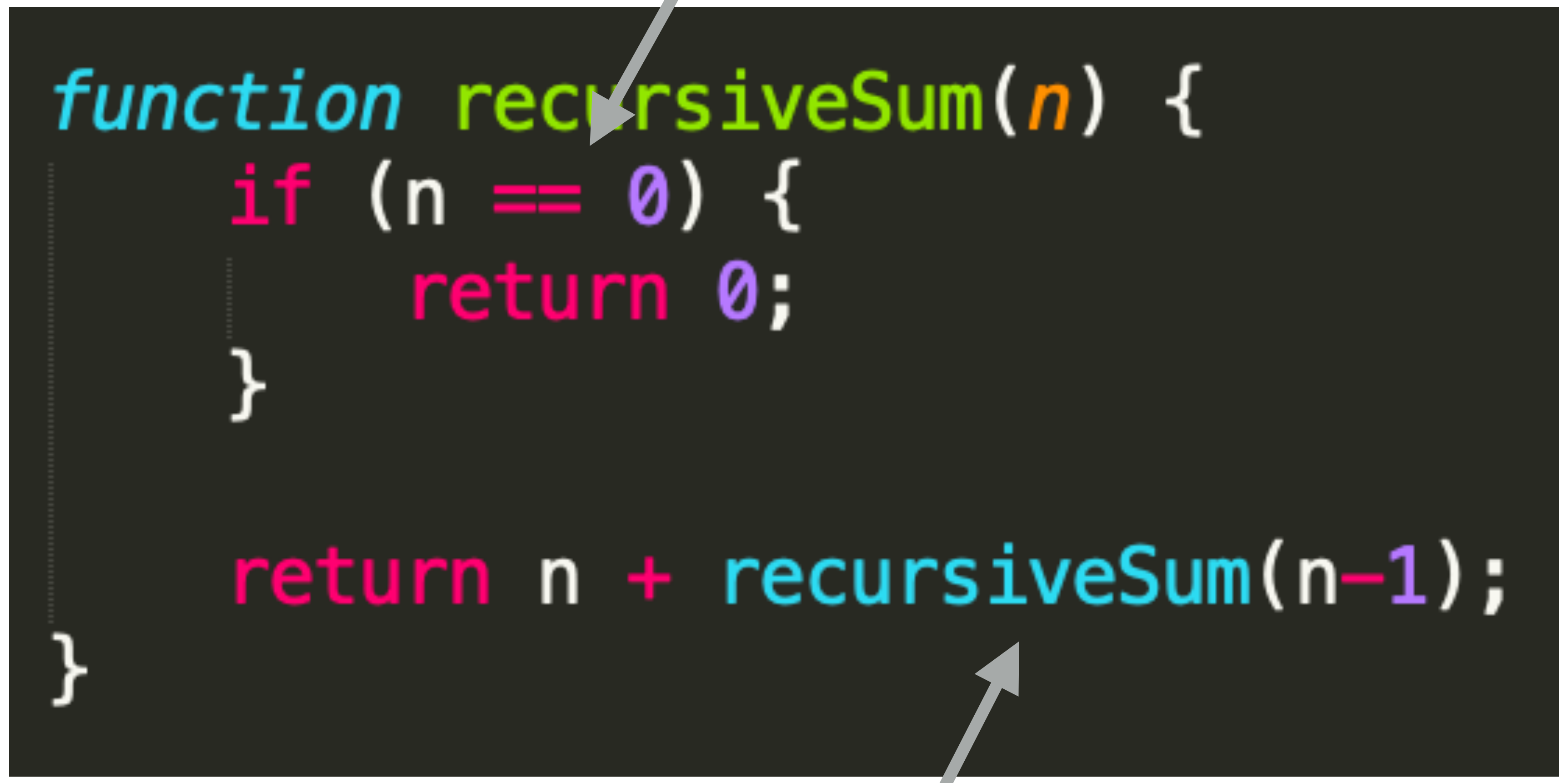
0 + 1 + 2 + 3 + … + (n-1) + n

Sum of all natural numbers up to and including n



```
function sumNatural(n) {
    var a = 0;

    if (n == 0) {
        return 0;
    }

    for (var i = 1; i <= n; i++) {
        a = a + i;
    }

    return a;
}
```

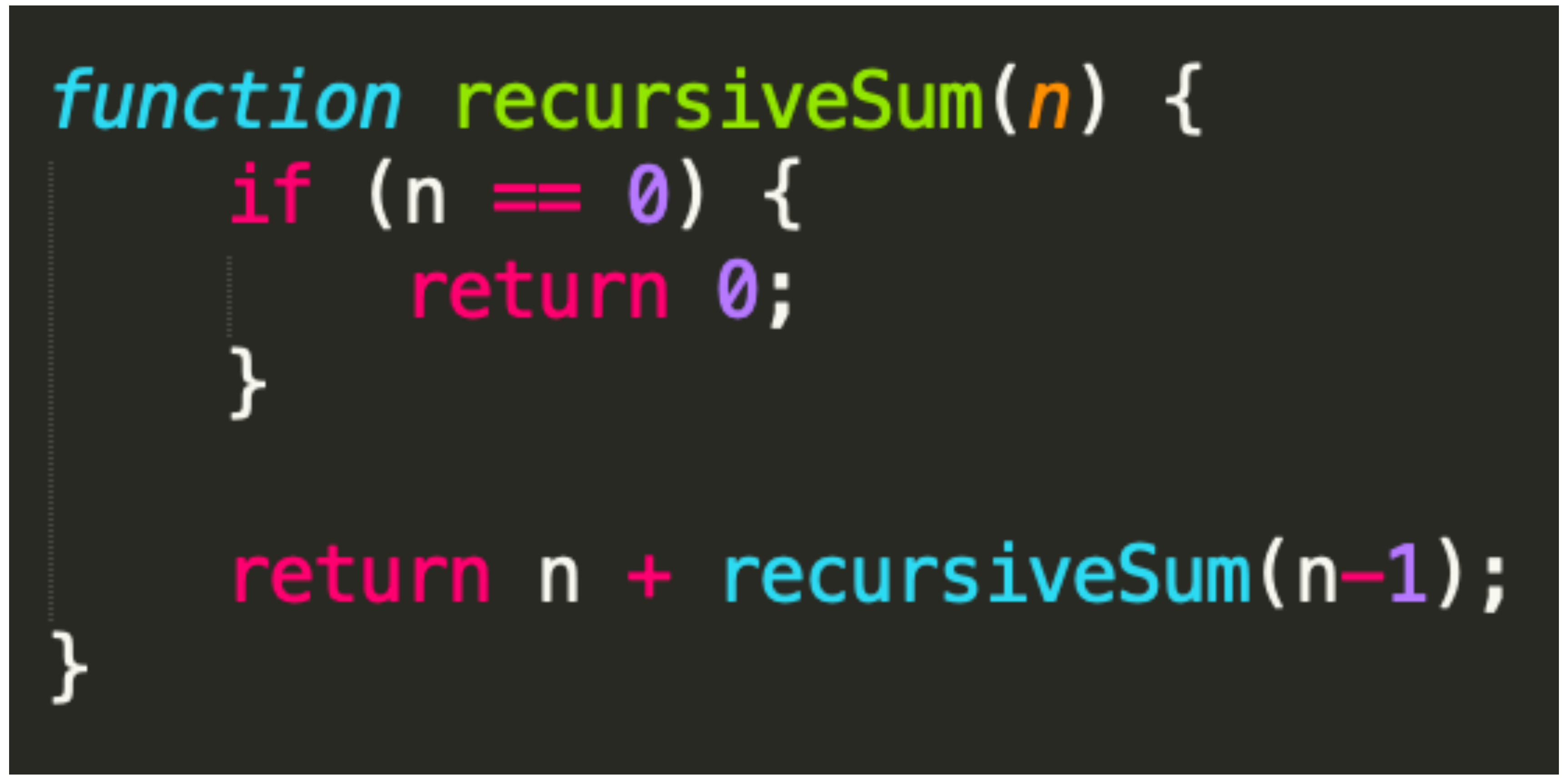
Base case of *n=0*



```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n-1);
}
```

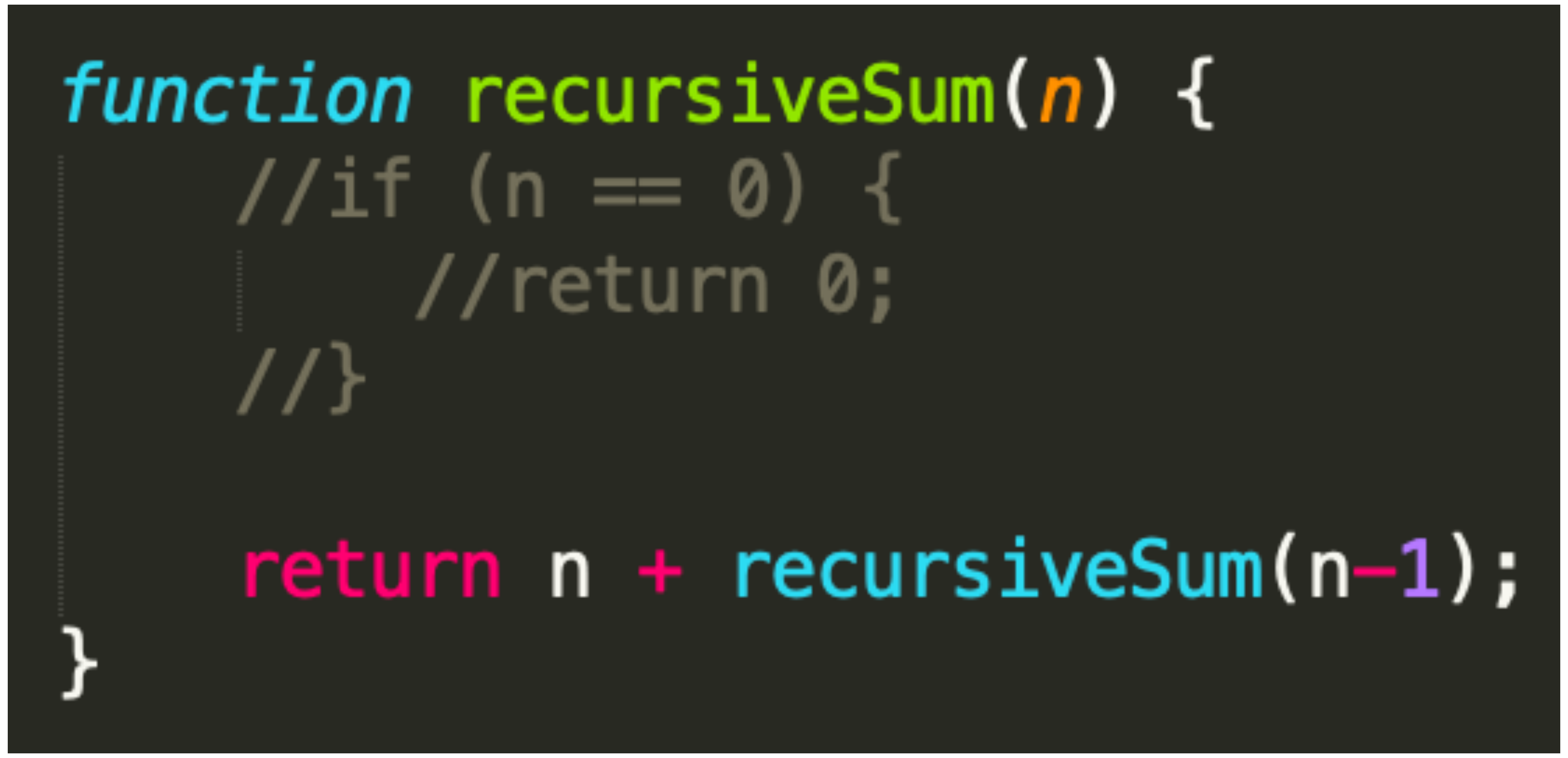
Function call on different input

Without base case, infinite loop

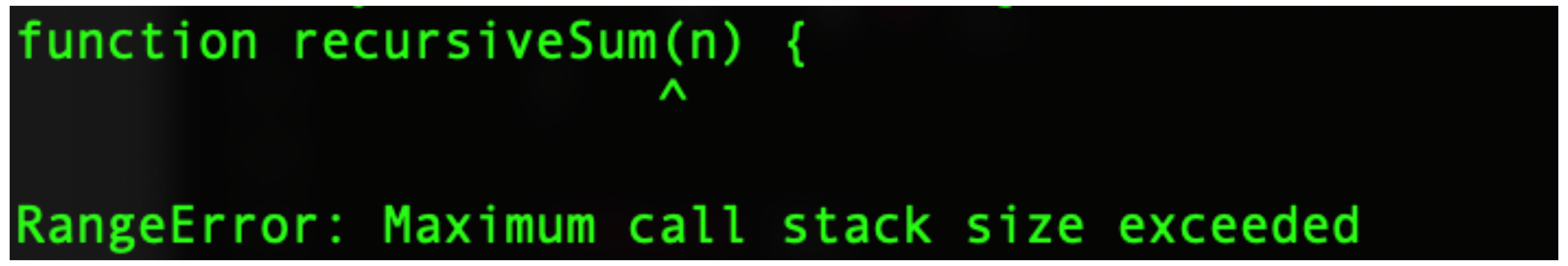


```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n-1);
}
```

Runs out of memory

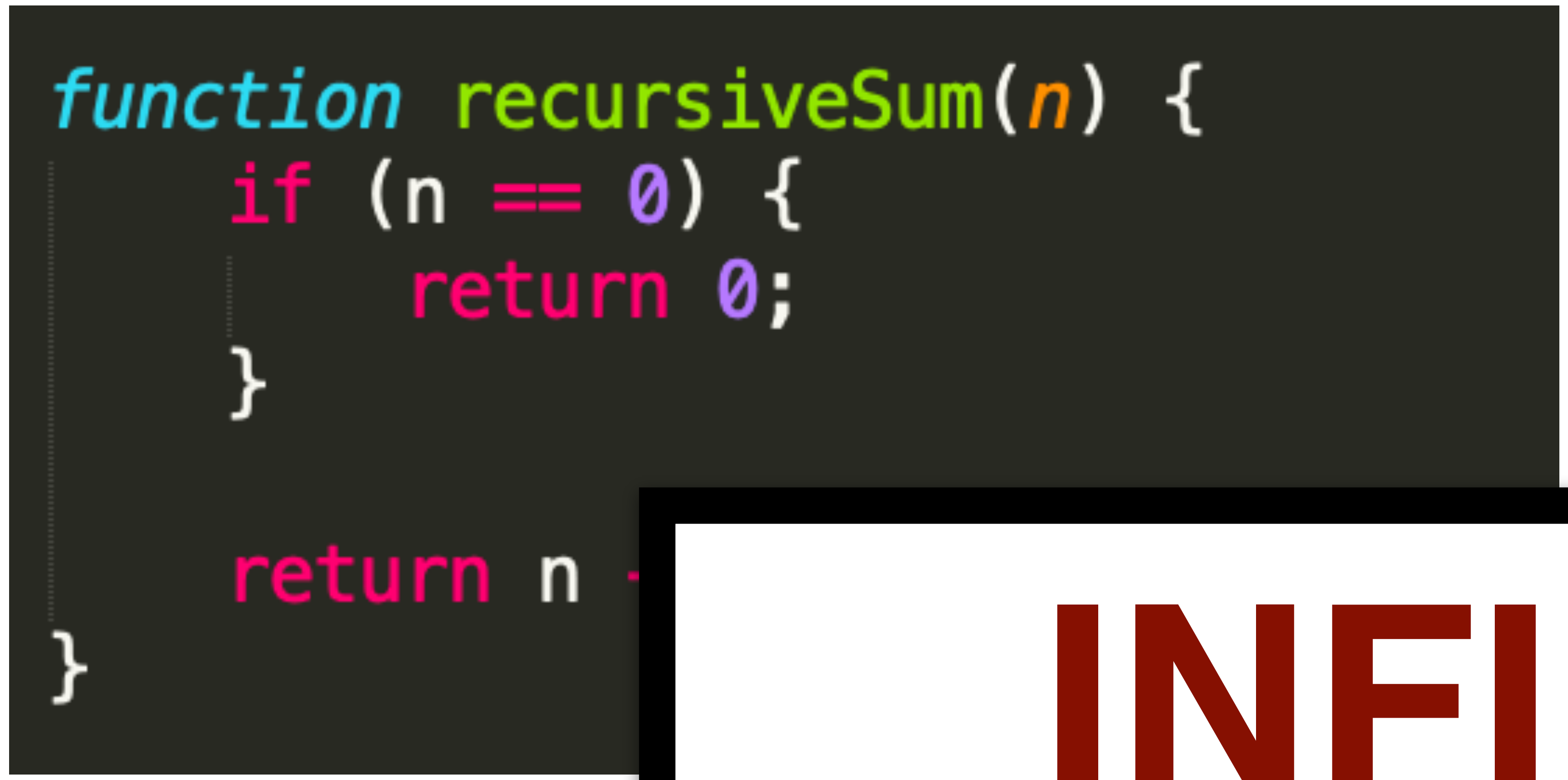


```
function recursiveSum(n) {
    //if (n == 0) {
        //return 0;
    //}

    return n + recursiveSum(n-1);
}
```



```
function recursiveSum(n) {
                     ^

RangeError: Maximum call stack size exceeded
```

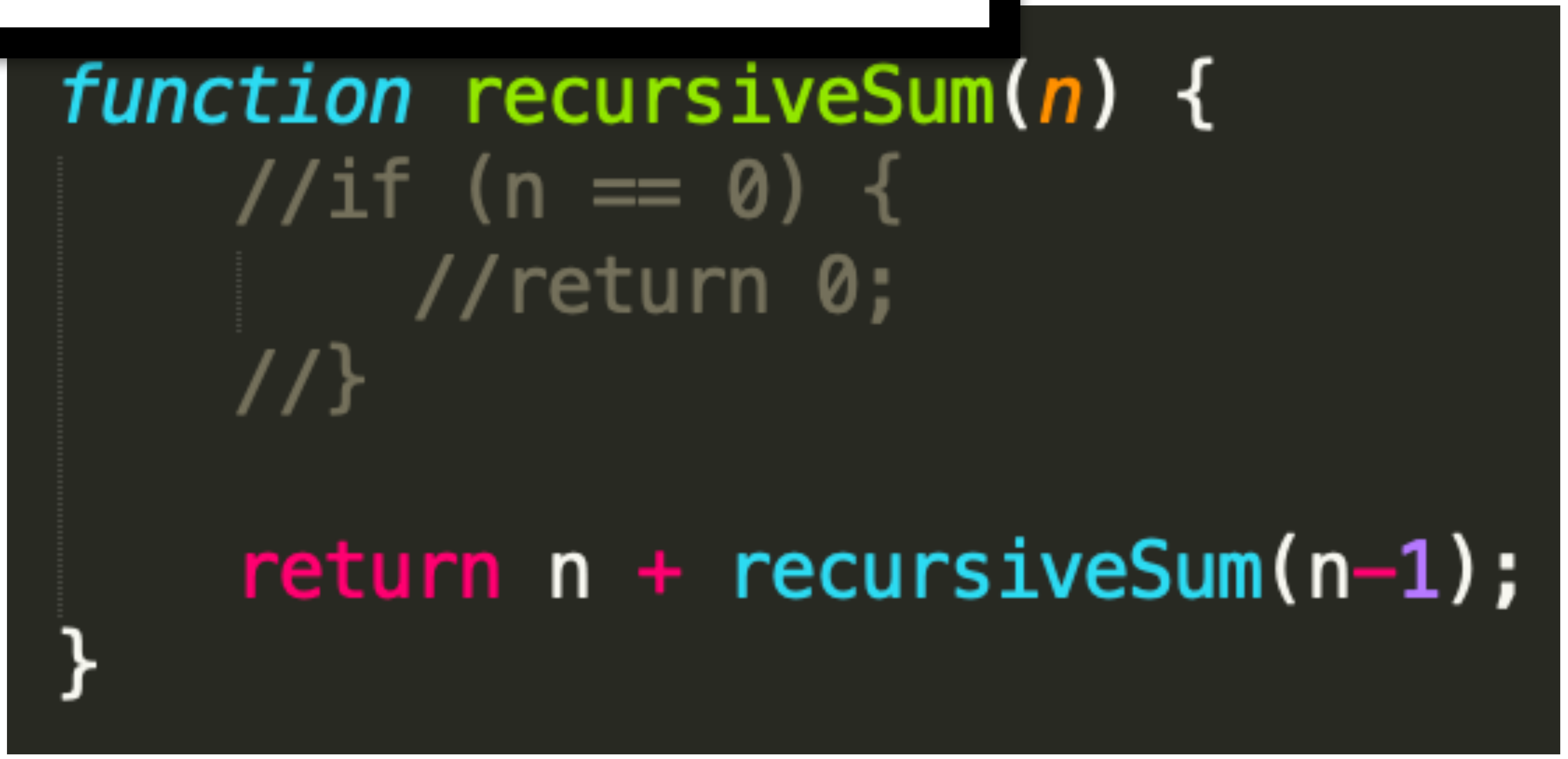
Without base case, infinite loop



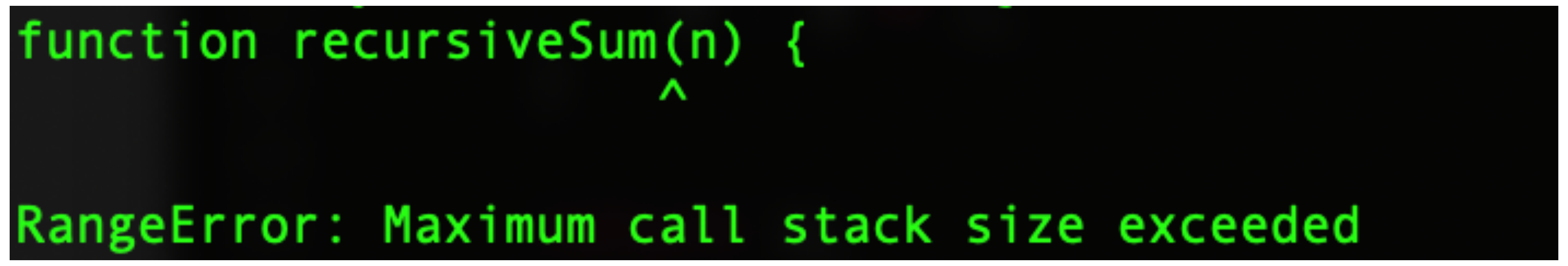
```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n
}
```

INFINITE RECURSION

nory

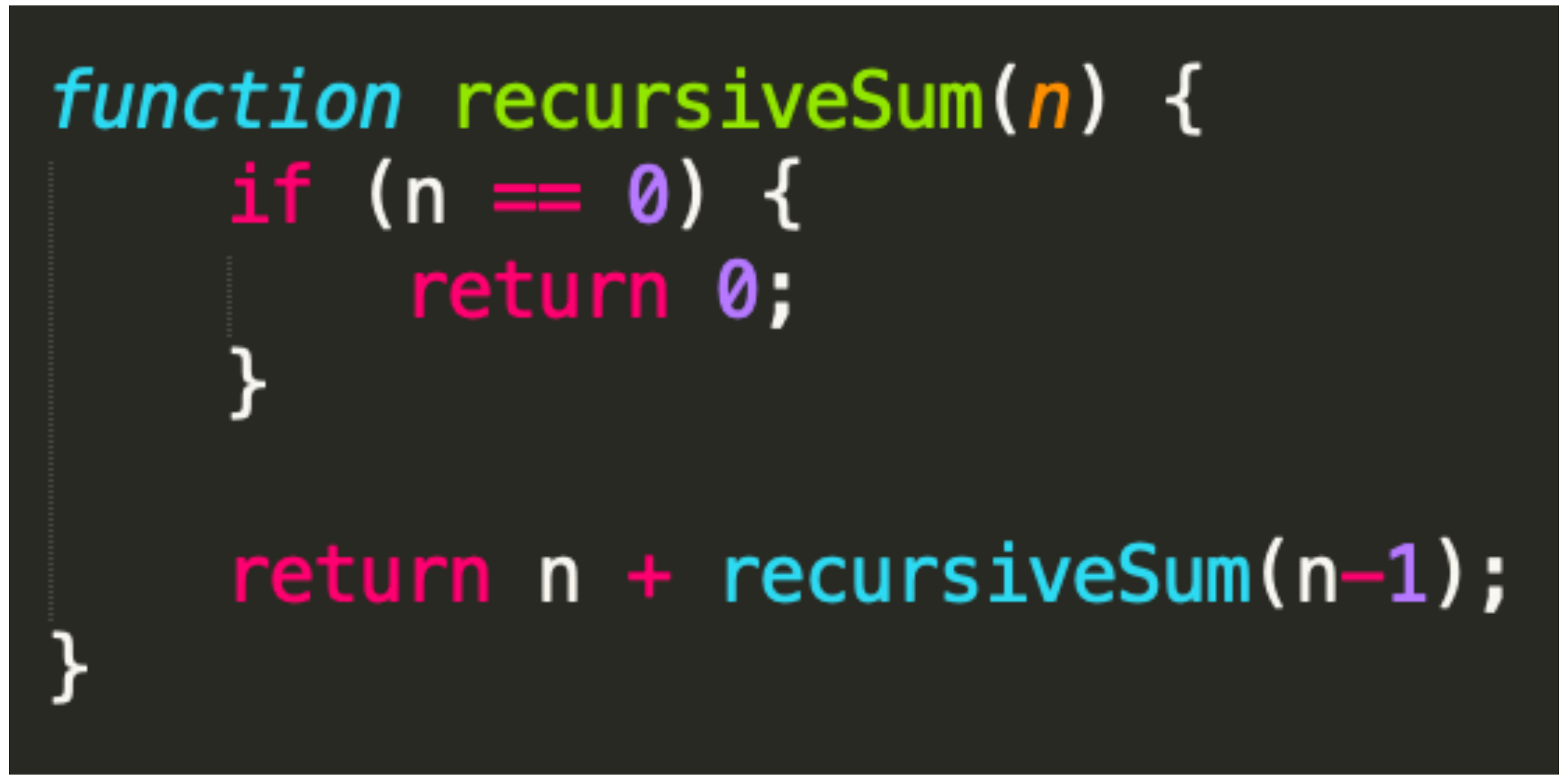


```
function recursiveSum(n) {
    //if (n == 0) {
        //return 0;
    //}

    return n + recursiveSum(n-1);
}
```



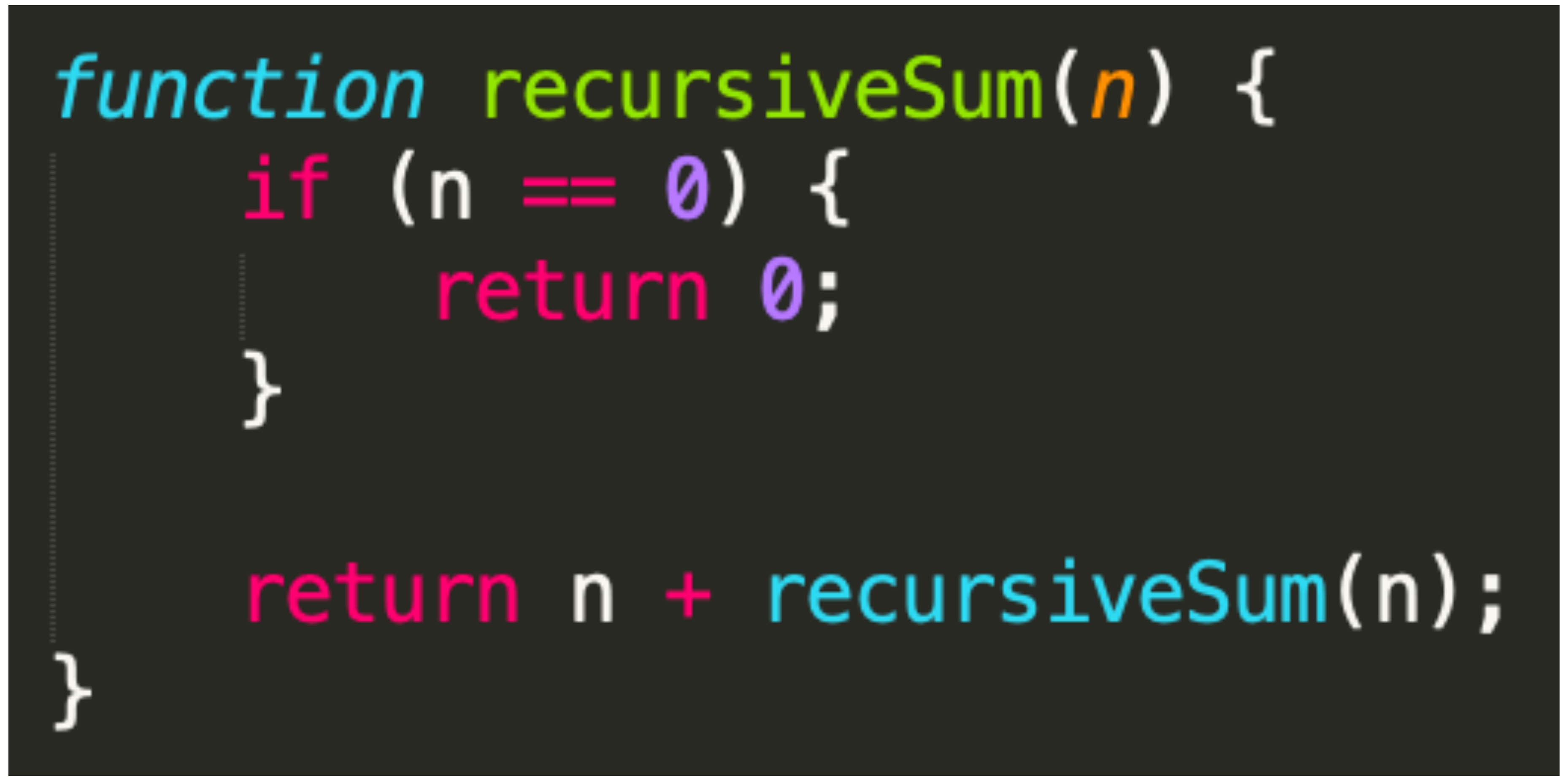
```
function recursiveSum(n) {
                        ^

RangeError: Maximum call stack size exceeded
```

Without altering argument in recursive call

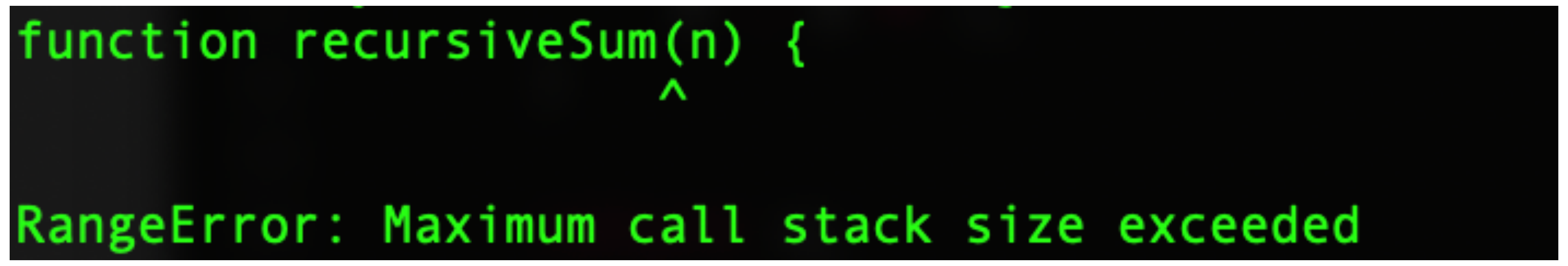


```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n-1);
}
```

Runs out of memory

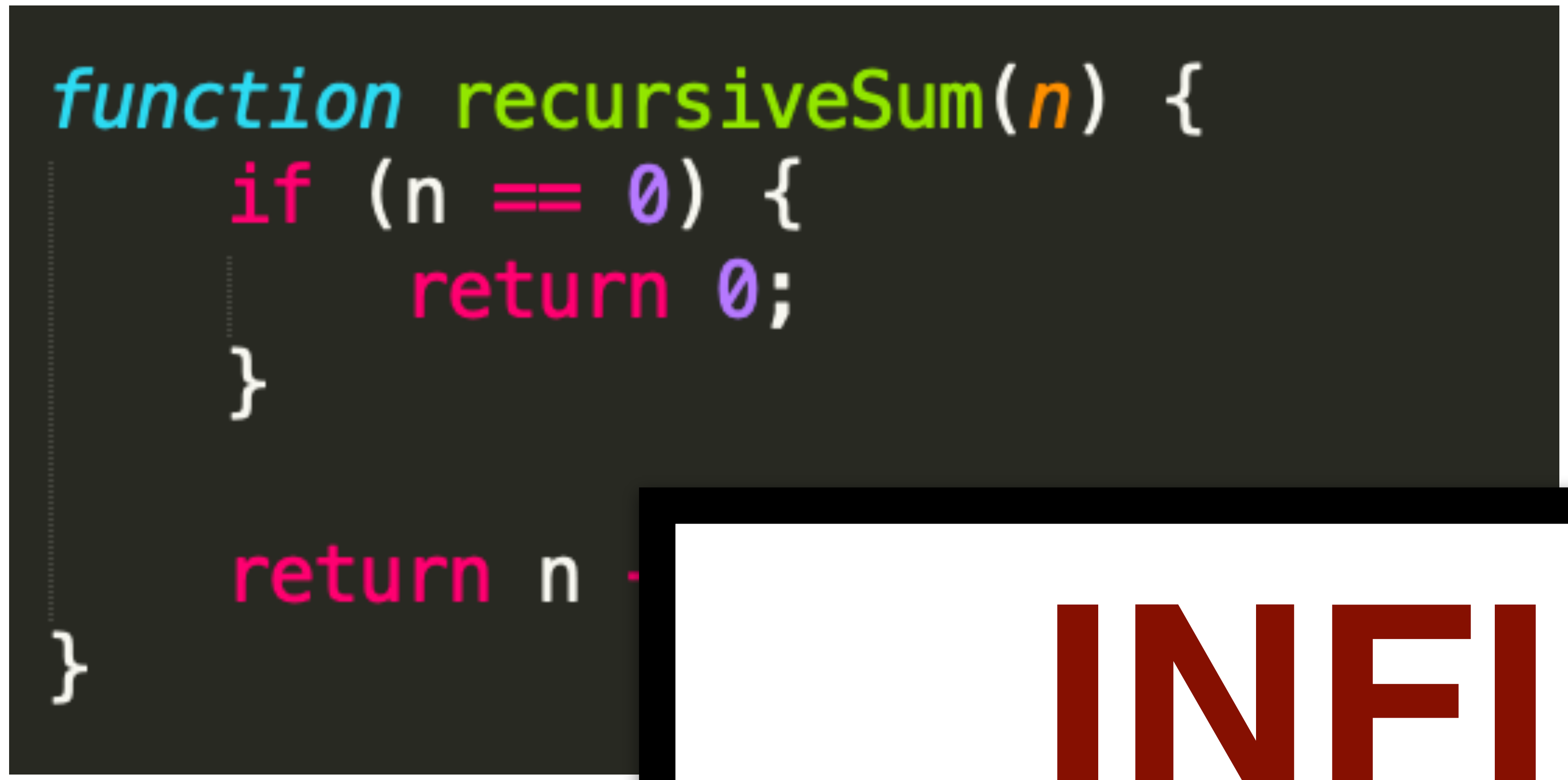


```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n);
}
```



```
function recursiveSum(n) {
                     ^

RangeError: Maximum call stack size exceeded
```

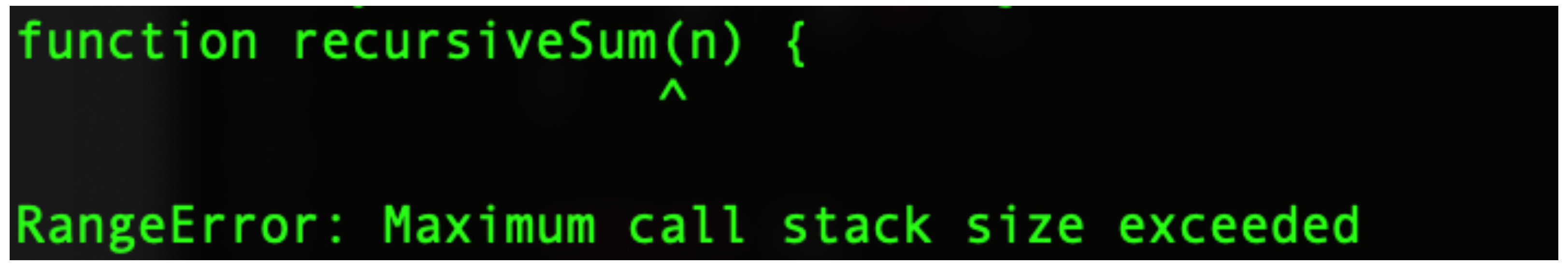
Without altering argument in recursive call



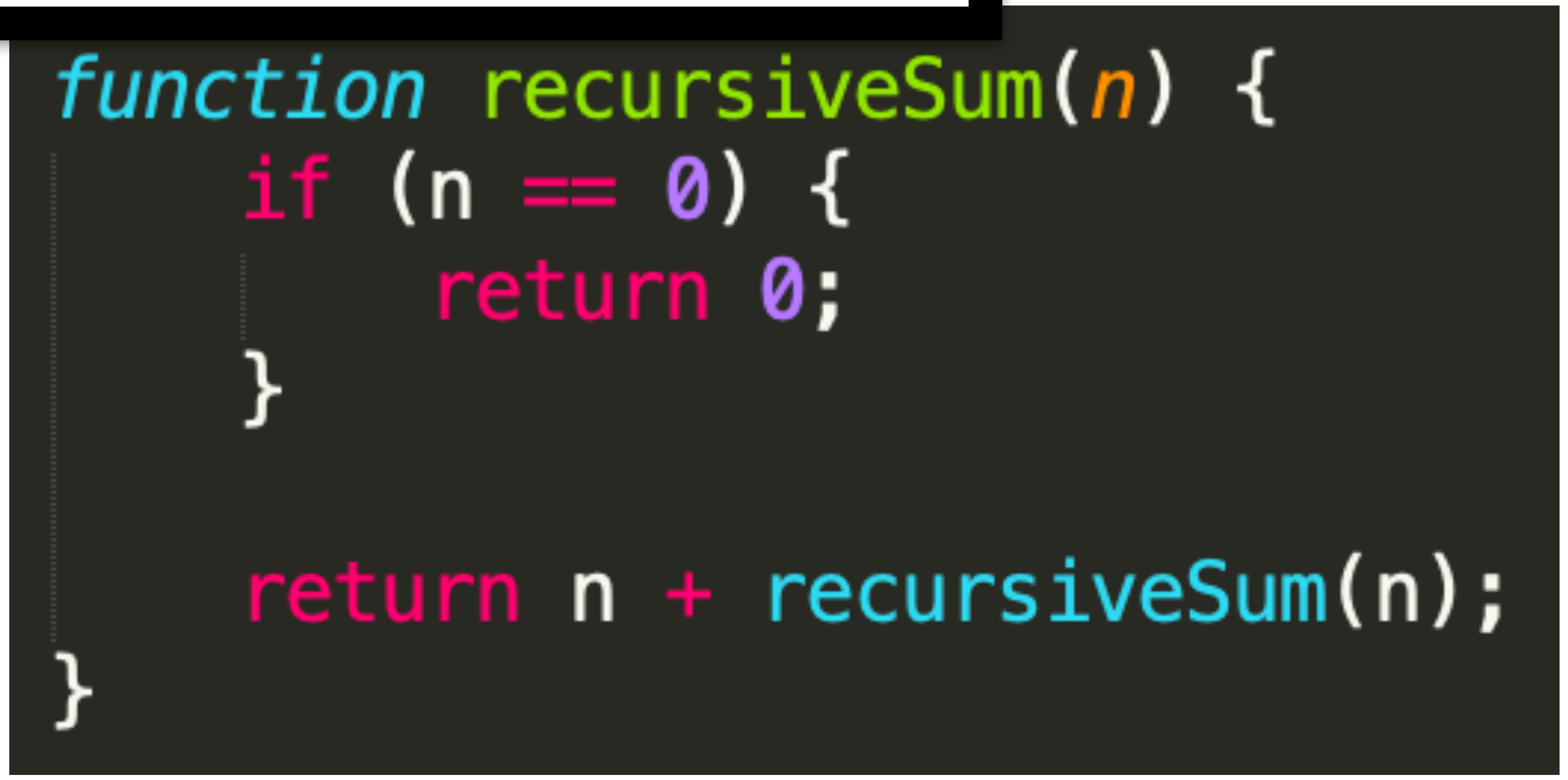
```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n
}
```

INFINITE RECURSION

nory



```
function recursiveSum(n) {
                        ^

RangeError: Maximum call stack size exceeded
```



```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n);
}
```

To end processes in the command line

CTRL + C

For recursion you will usually get a runtime error before needing CTRL + C (stack overflow - more on this later)

For infinite (while) loops this will stop things

Admin

- Seventh and final quiz available today at 4pm
  - Fifth quiz deadline today at 4pm
  - Sixth quiz deadline next Monday at 4pm

- Sudoku assignment due today **15th March 4pm**

- Primes assignment
  - Deadline **15th March 4pm**
  - Cut-off date **29th March 4pm**
  - Help with Primes Assignment this week in VCH

- No new worksheet this week - new worksheet next week
  - Consider the exercises mentioned in the lecture if you want more

- Mock online test available next Monday

Let’s practice some recursion

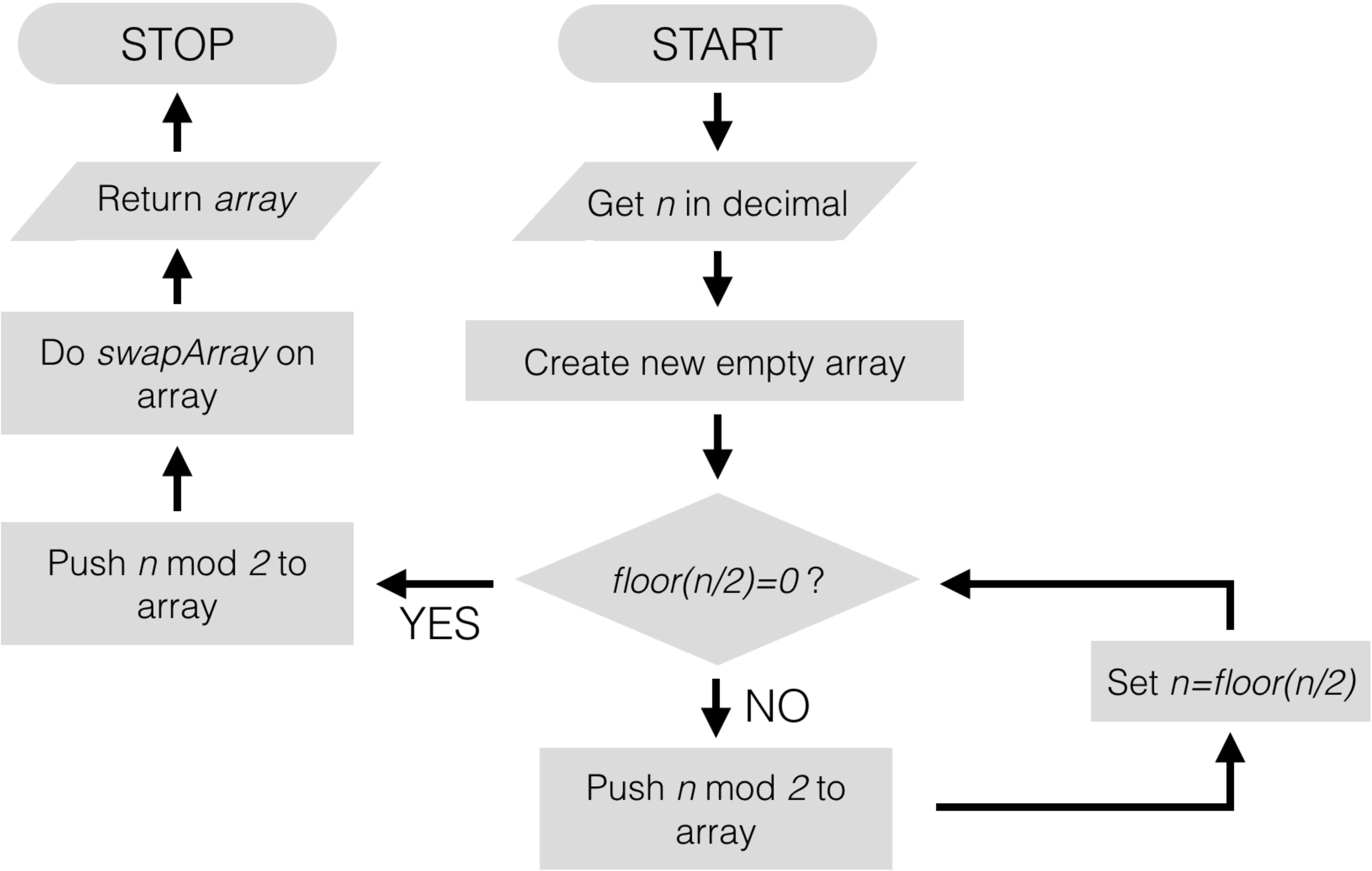
Recursion.zip in Lecture 9 Resources on learn.gold

Recursive functions for:

Factorial of *n*: the product of all numbers from 1 to n
n!=n(n-1)(n-2)…1

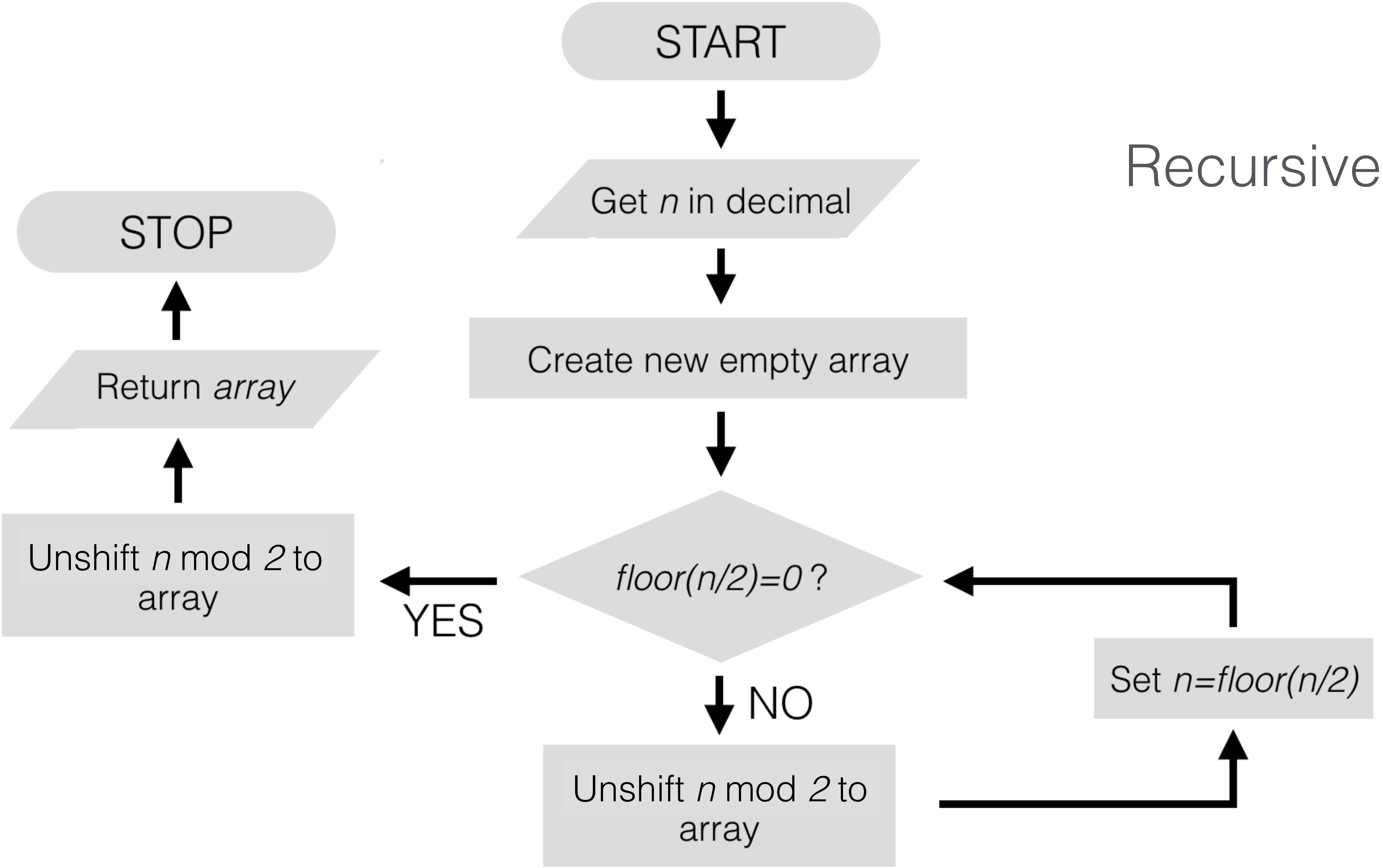
Nth Fibonacci number: the sum of the *(N-1)th* and *(N-2)th* Fibonacci numbers starting with
N=0: 0
N=1: 1
N=2: 1
N=3: 2
N=4: 3
…

Worksheet 1

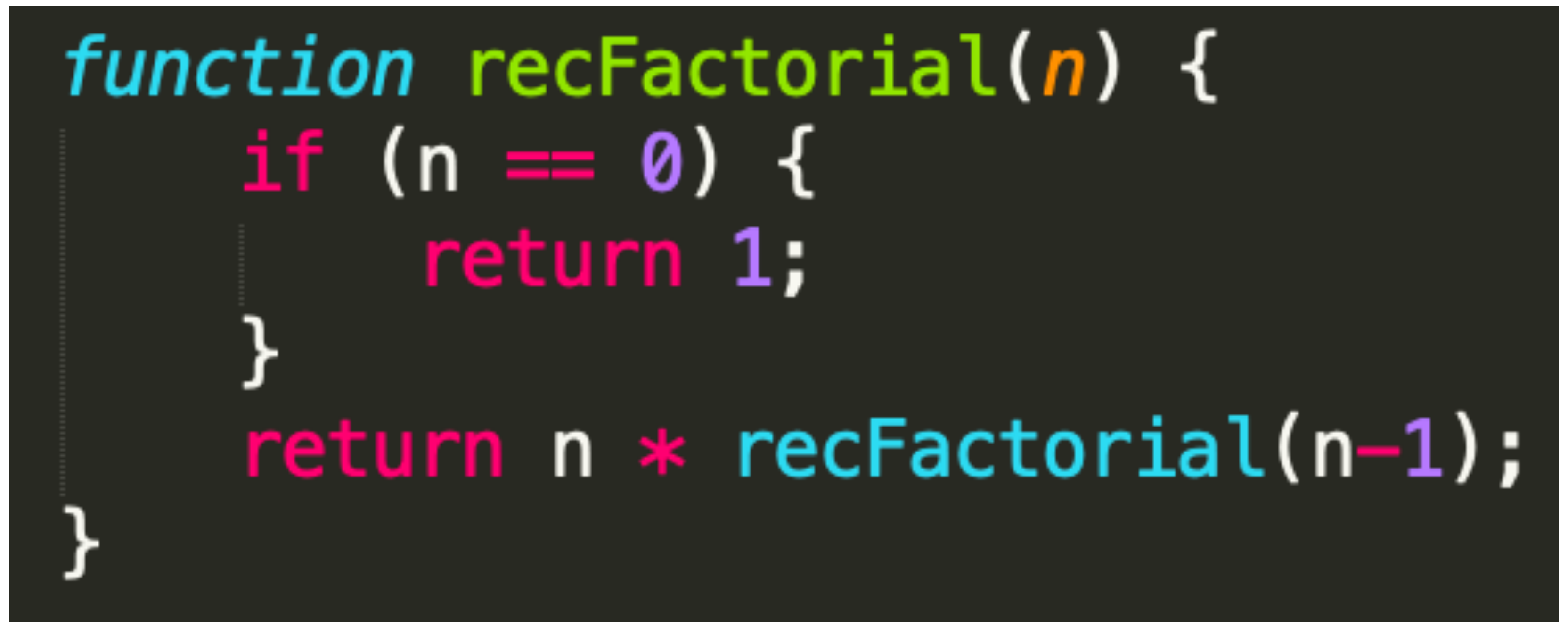

STOP
START
Return *array*
Get *n* in decimal
Do *swapArray* on array
Create new empty array
Push *n* mod *2* to array
YES
floor(n/2)=0 ?
NO
Set *n=floor(n/2)*
Push *n* mod *2* to array

Simplified Worksheet 1

Recursive function

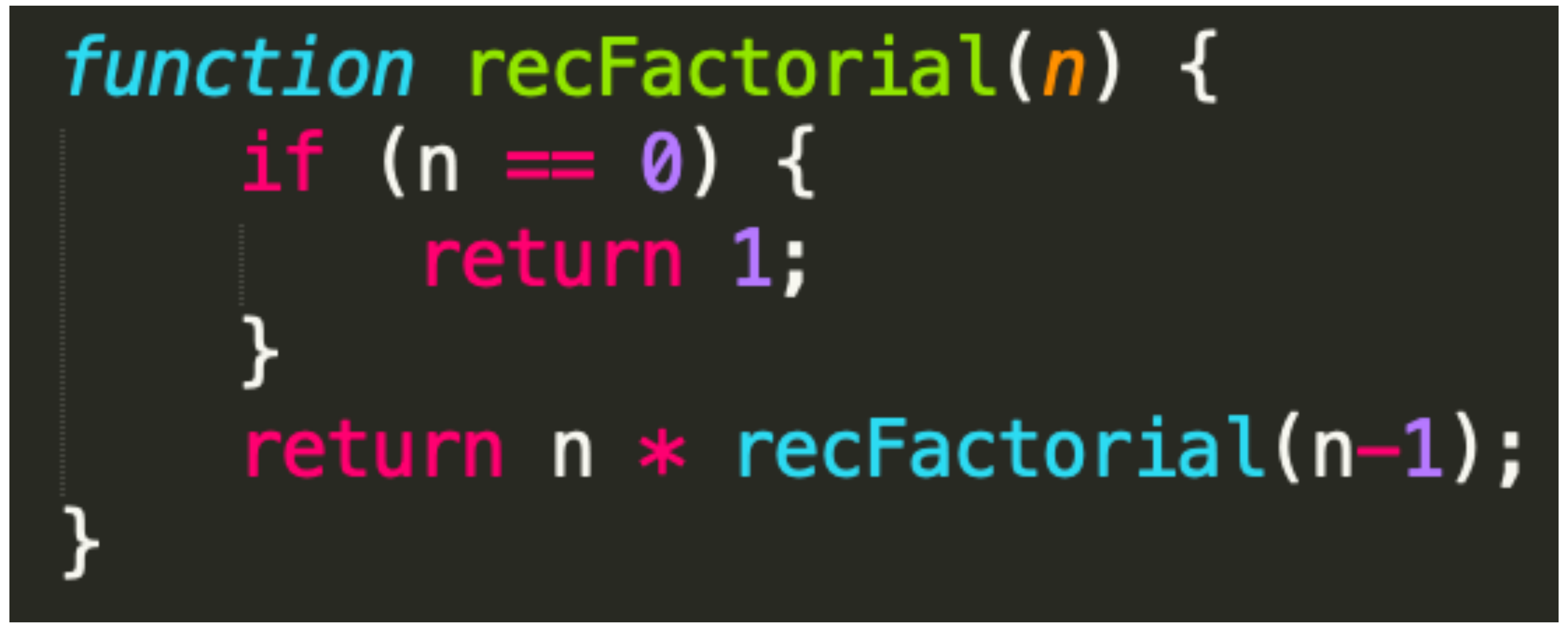


Recursion and Call Stacks



```
function recFactorial(n) {
    if (n == 0) {
        return 1;
    }
    return n * recFactorial(n-1);
}
```

How does your computer know what to do?

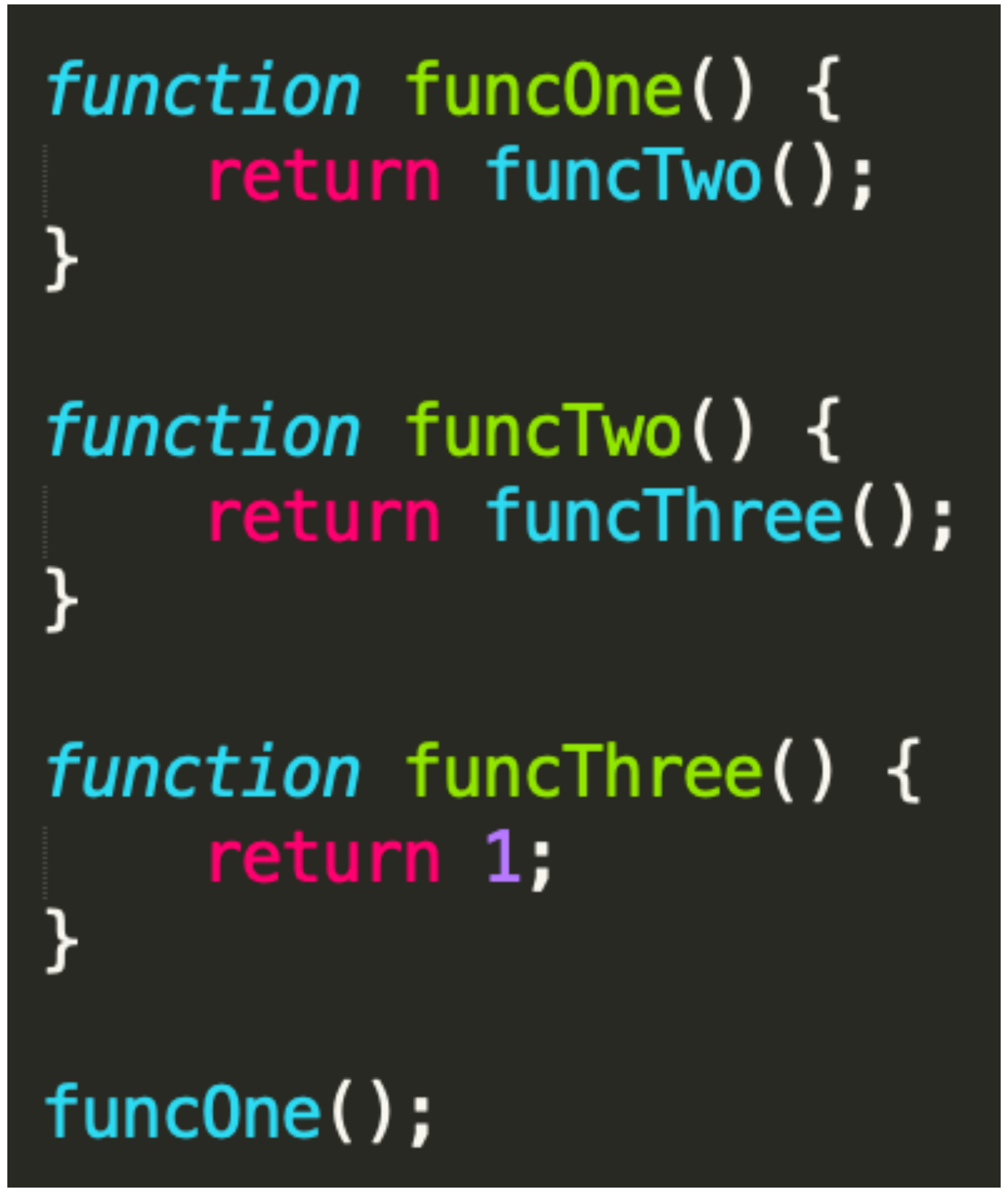


```
function recFactorial(n) {
    if (n == 0) {
        return 1;
    }
    return n * recFactorial(n-1);
}
```

How does your computer know what to do?

It uses a stack

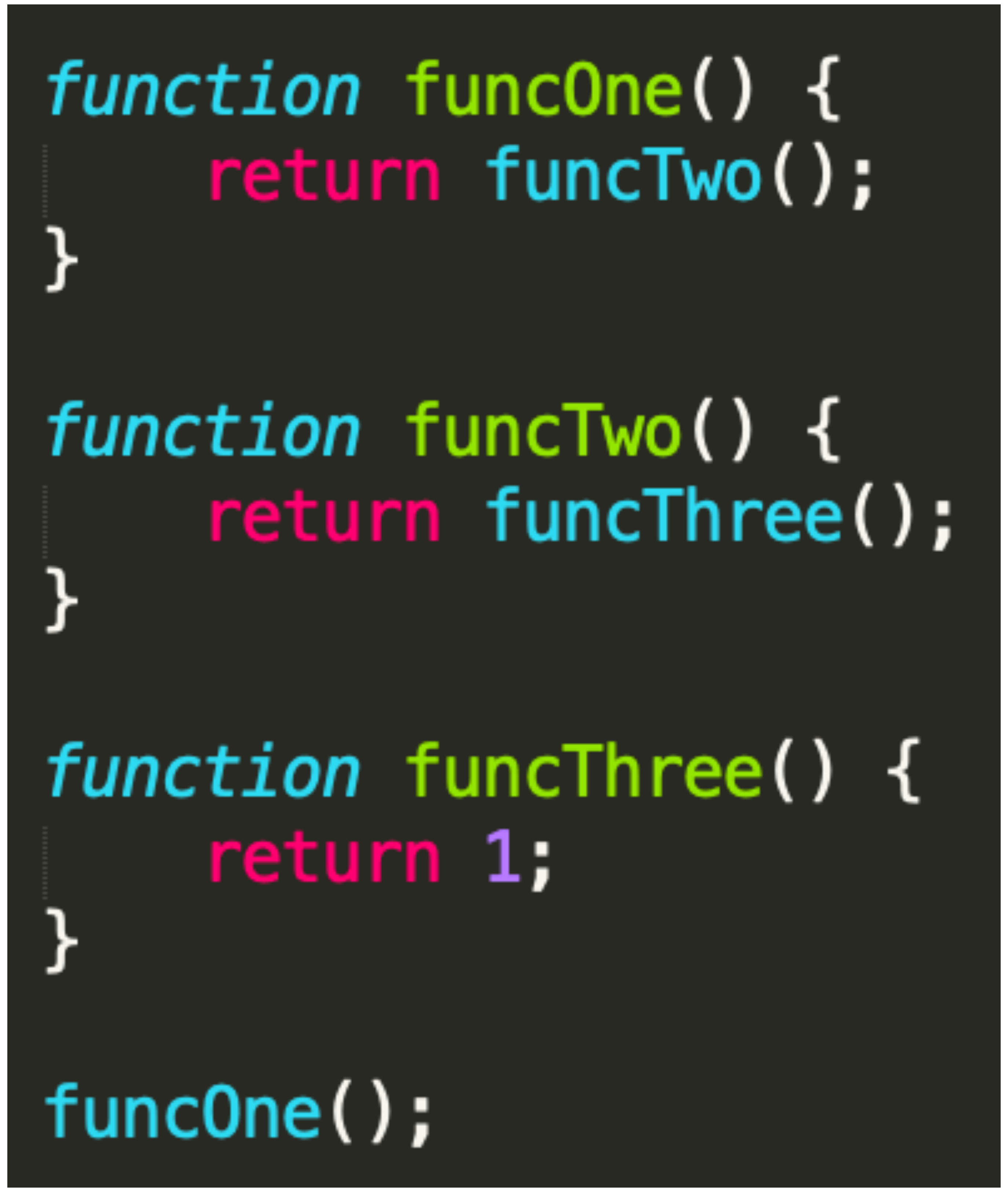
Call stack: simplified picture



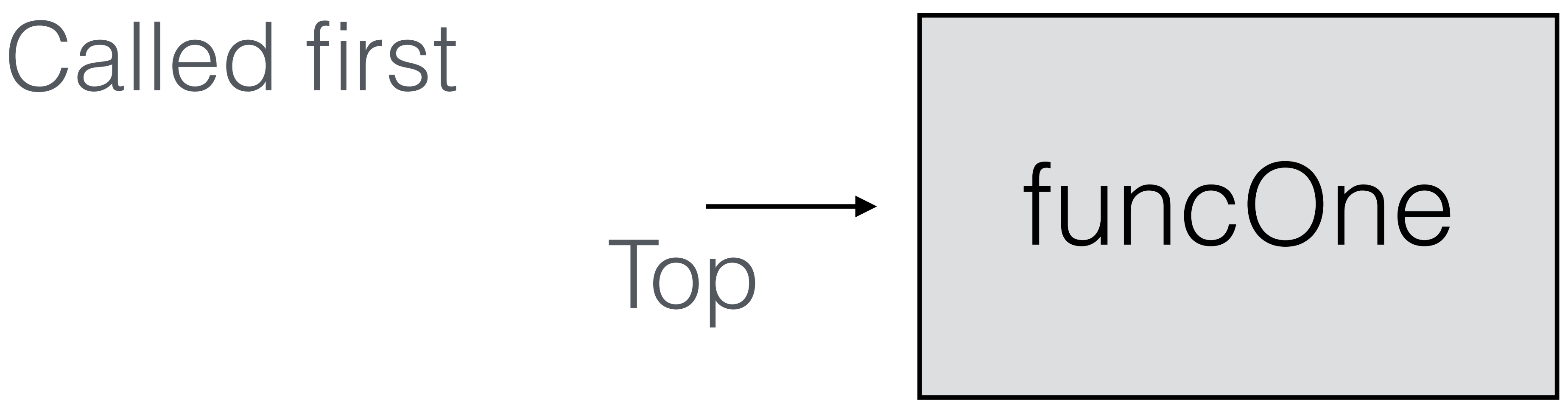
```
function funcOne() {
    return funcTwo();
}

function funcTwo() {
    return funcThree();
}

function funcThree() {
    return 1;
}

funcOne();
```

Empty stack

Call stack: simplified picture



```
function funcOne() {
    return funcTwo();
}

function funcTwo() {
    return funcThree();
}

function funcThree() {
    return 1;
}

funcOne();
```

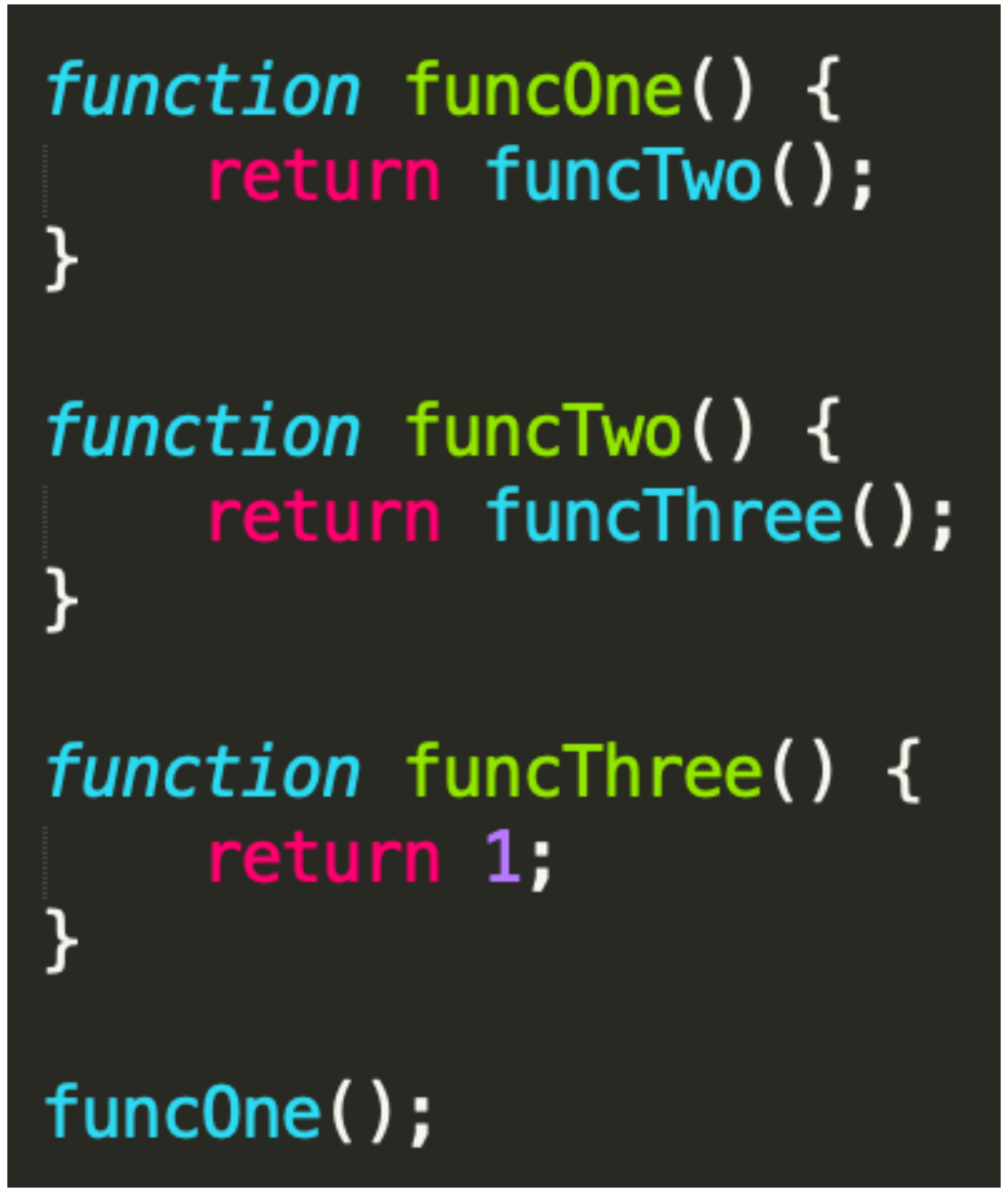


Called first

Top

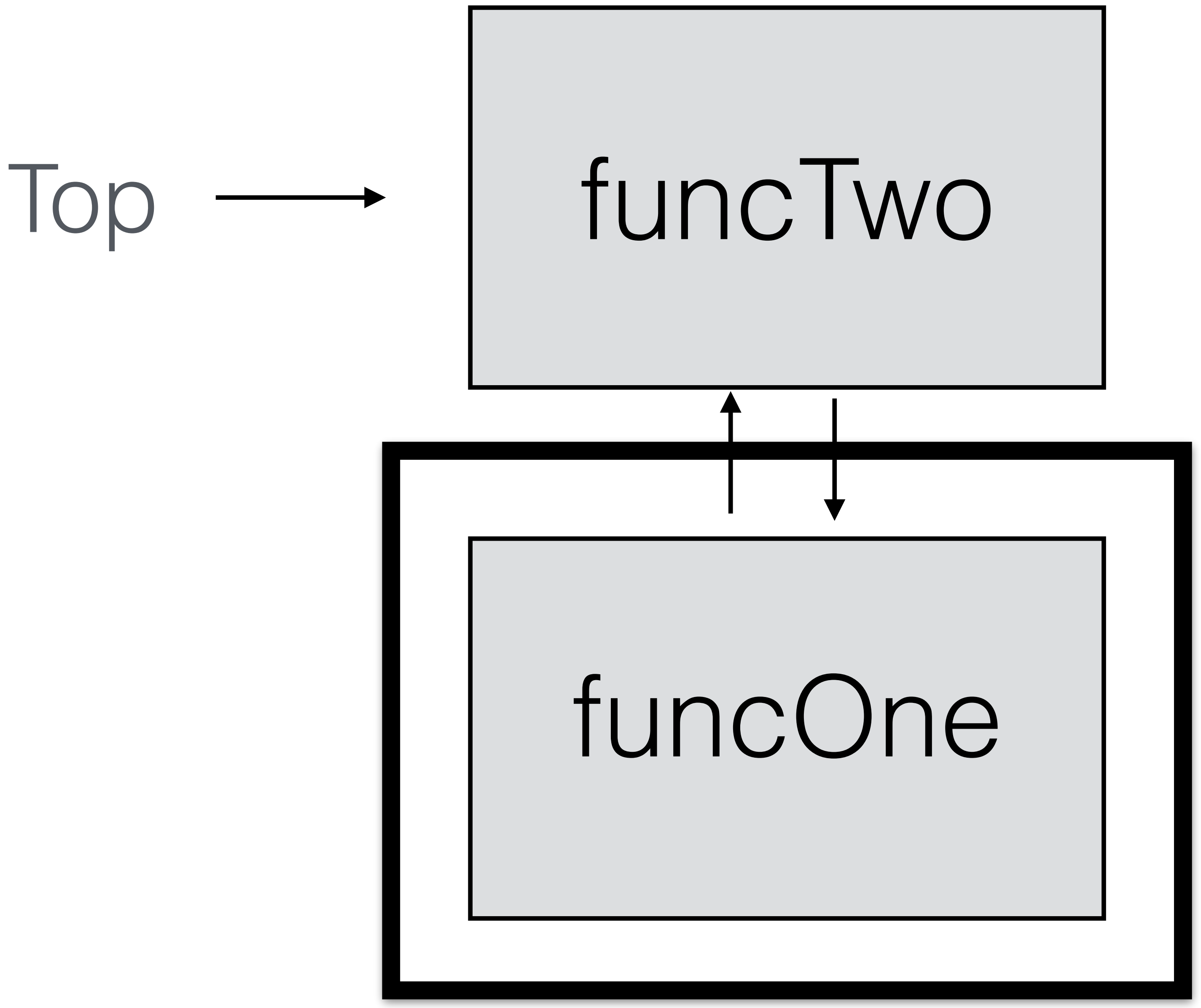
funcOne

Call stack: simplified picture

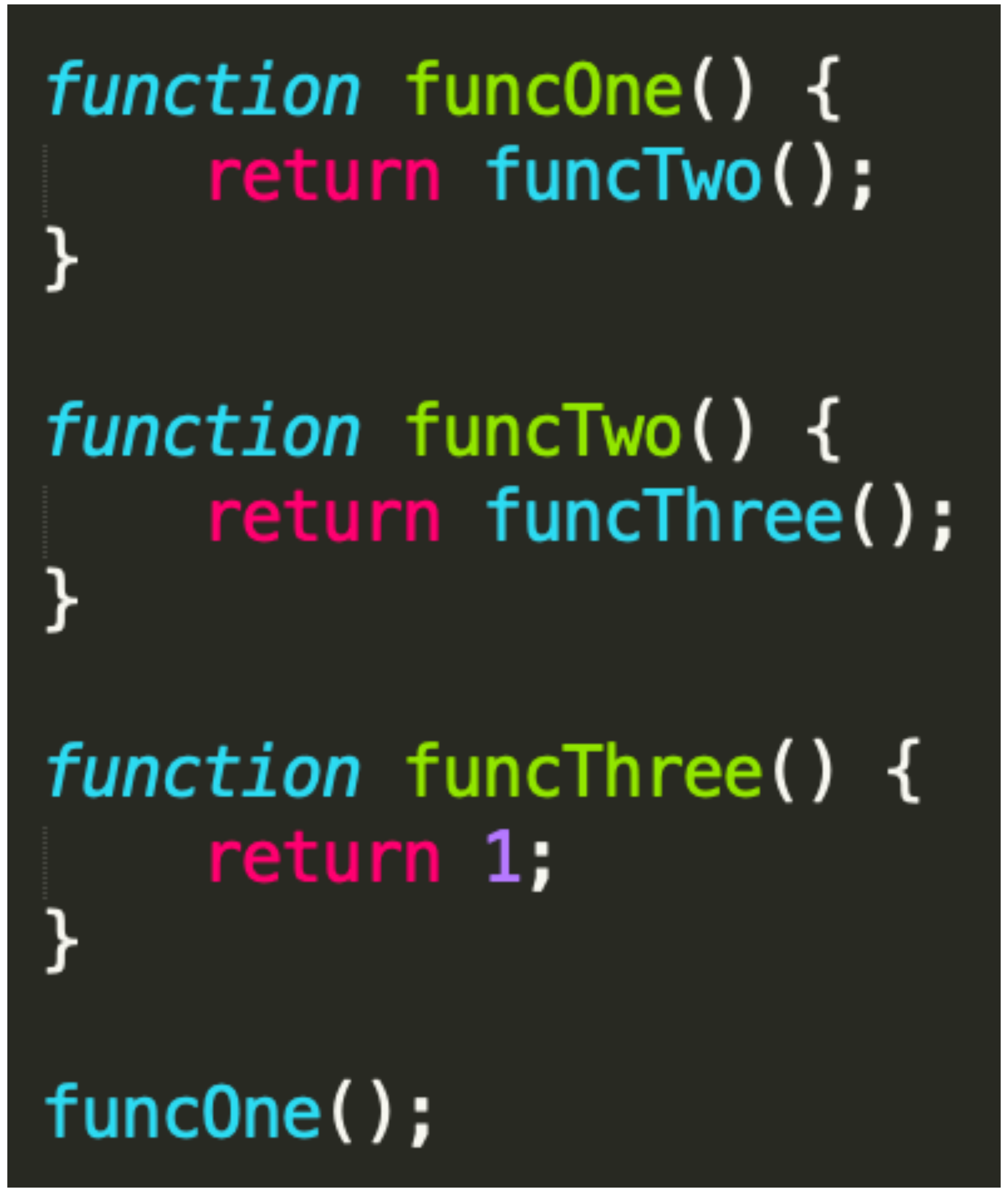


```
function funcOne() {
    return funcTwo();
}

function funcTwo() {
    return funcThree();
}

function funcThree() {
    return 1;
}

funcOne();
```

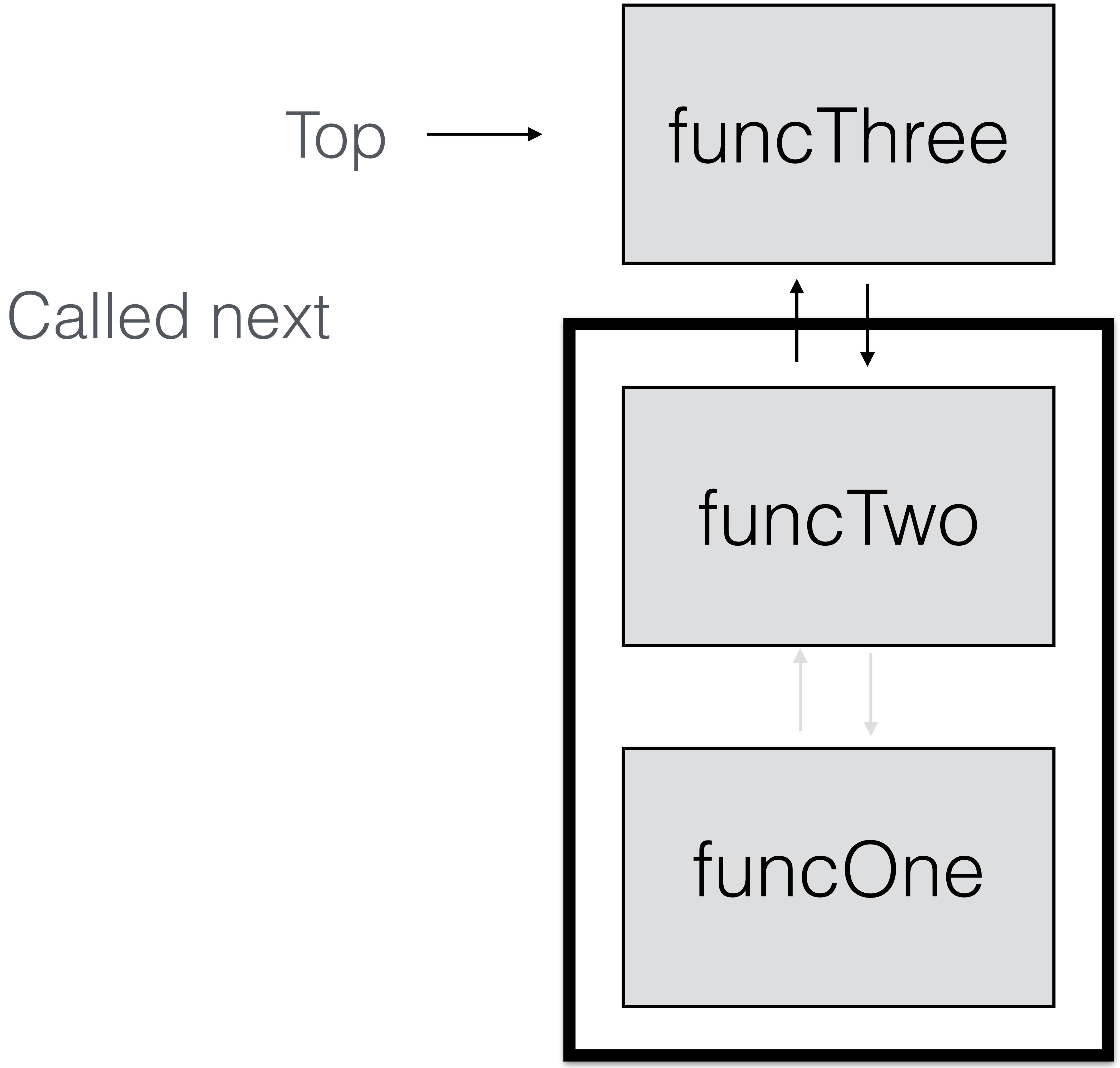
Called next



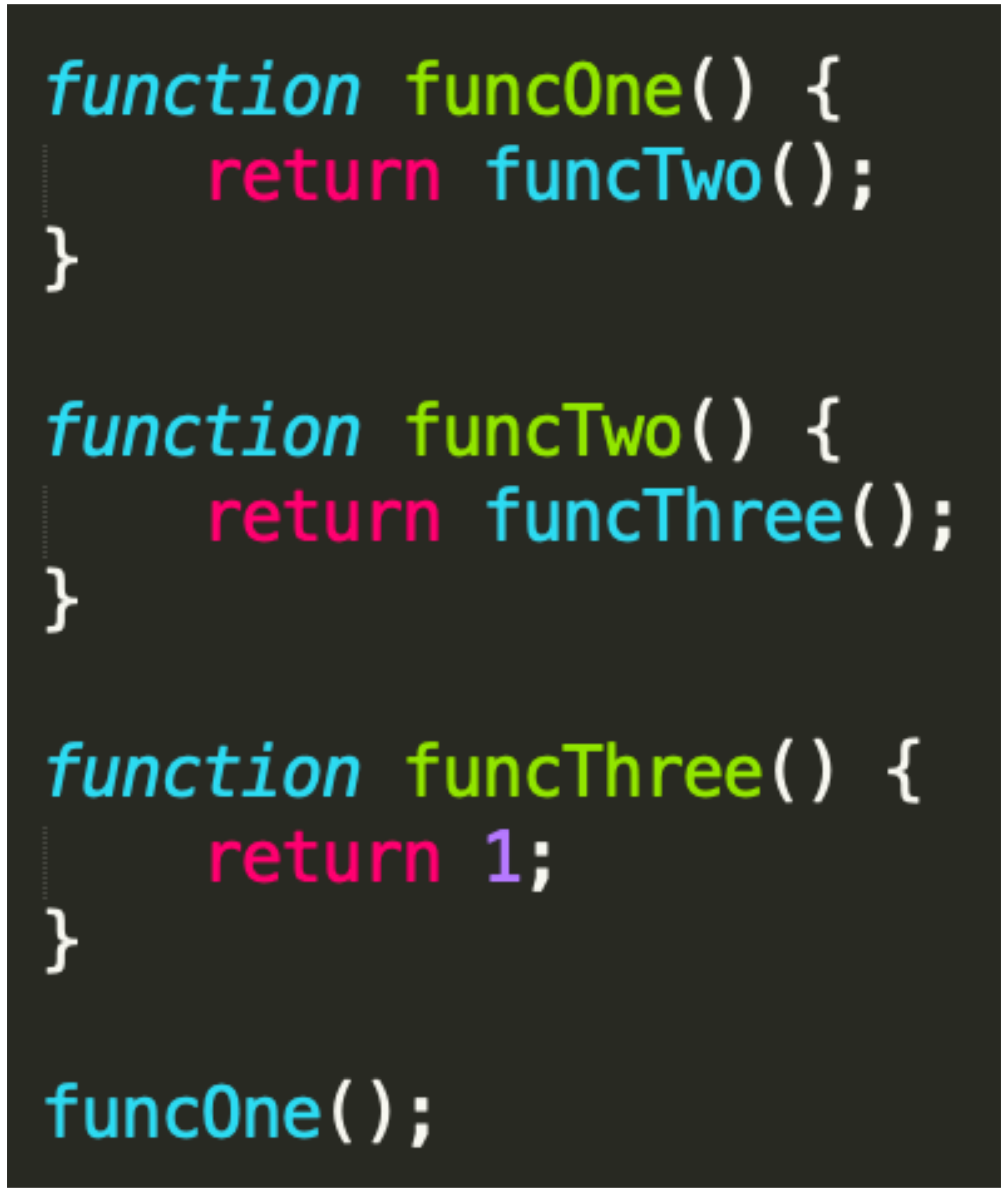
Call stack: simplified picture



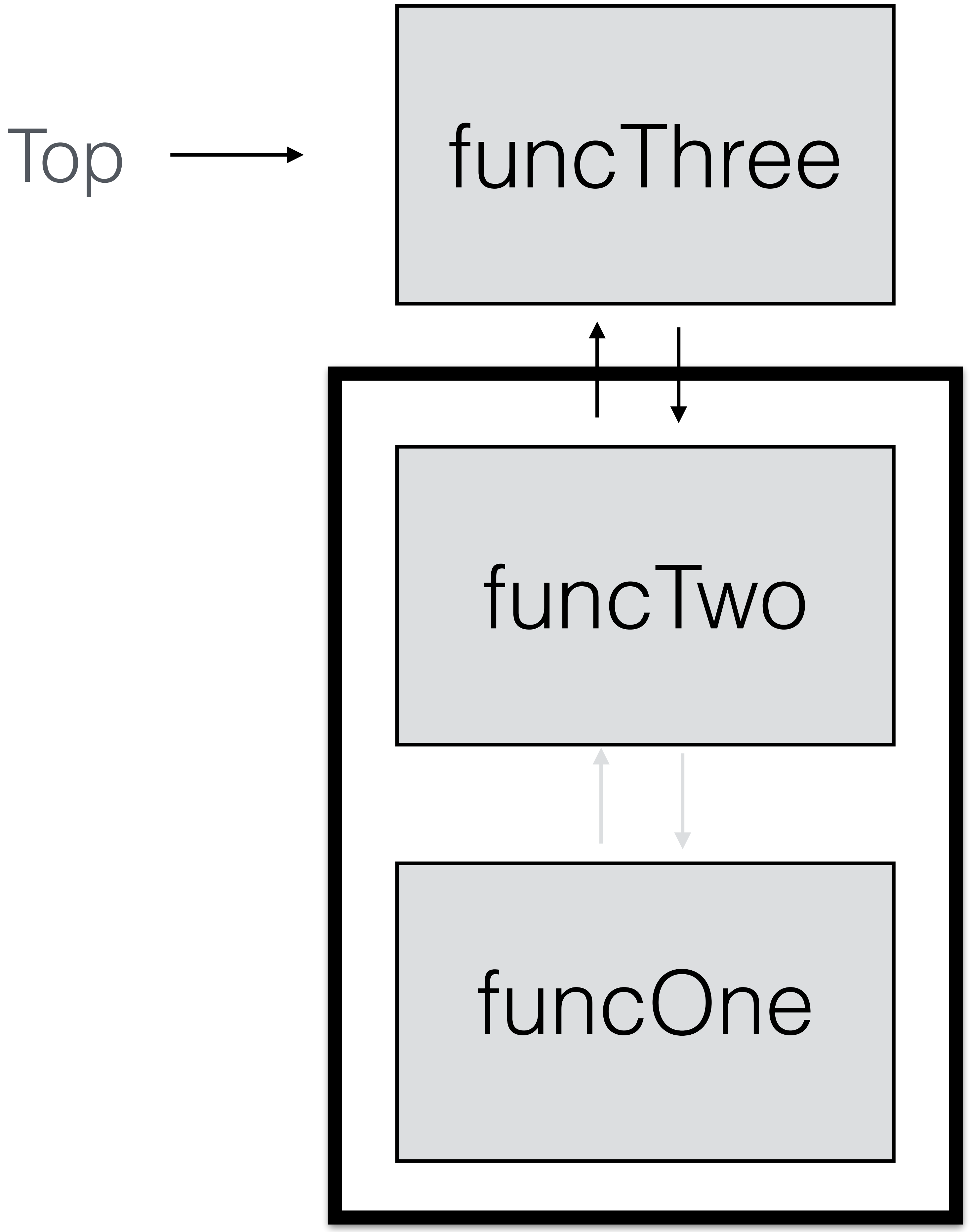
```
function funcOne() {
    return funcTwo();
}

function funcTwo() {
    return funcThree();
}

function funcThree() {
    return 1;
}

funcOne();
```



Call stack: simplified picture

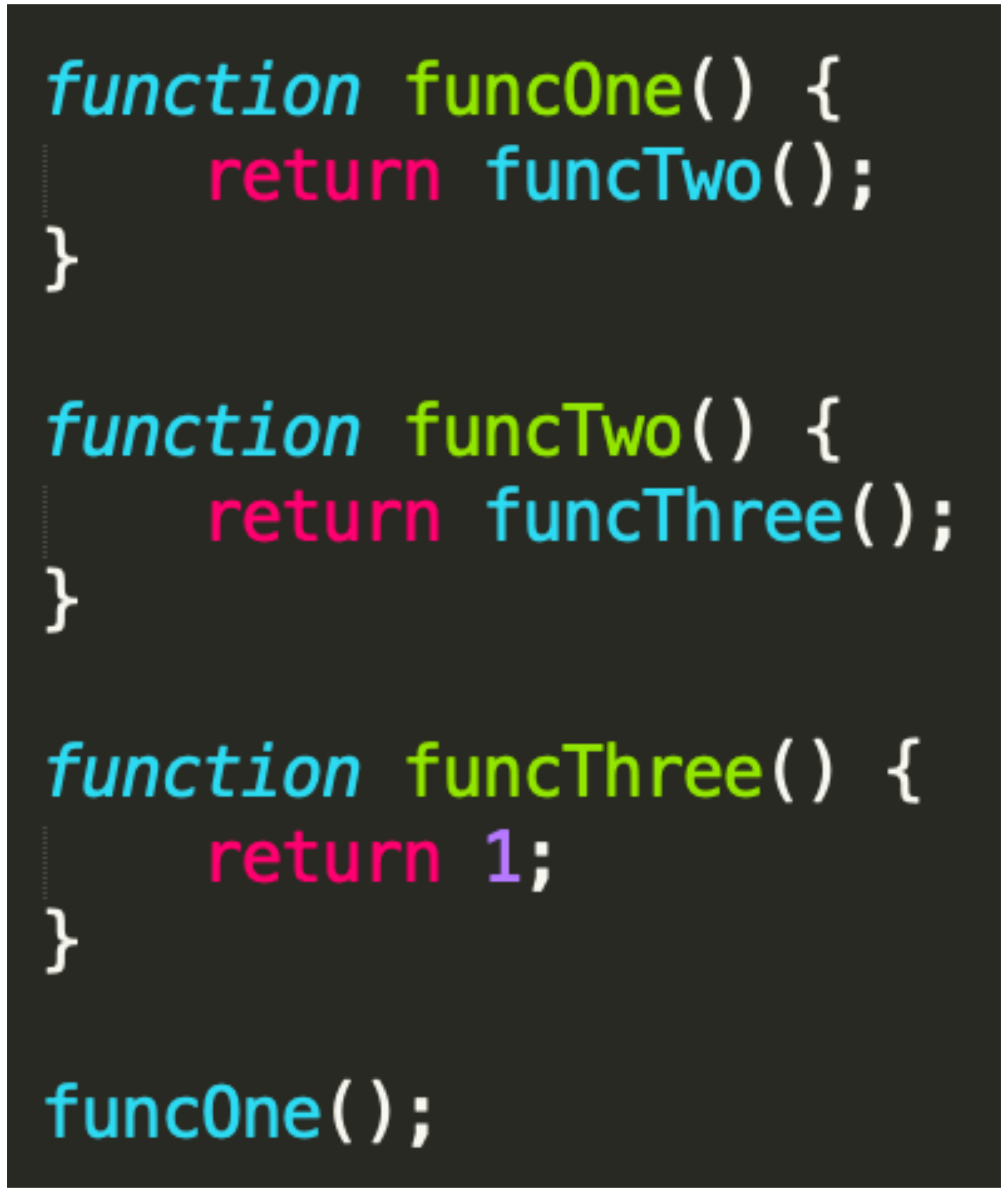


```
function funcOne() {
    return funcTwo();
}

function funcTwo() {
    return funcThree();
}

function funcThree() {
    return 1;
}

funcOne();
```



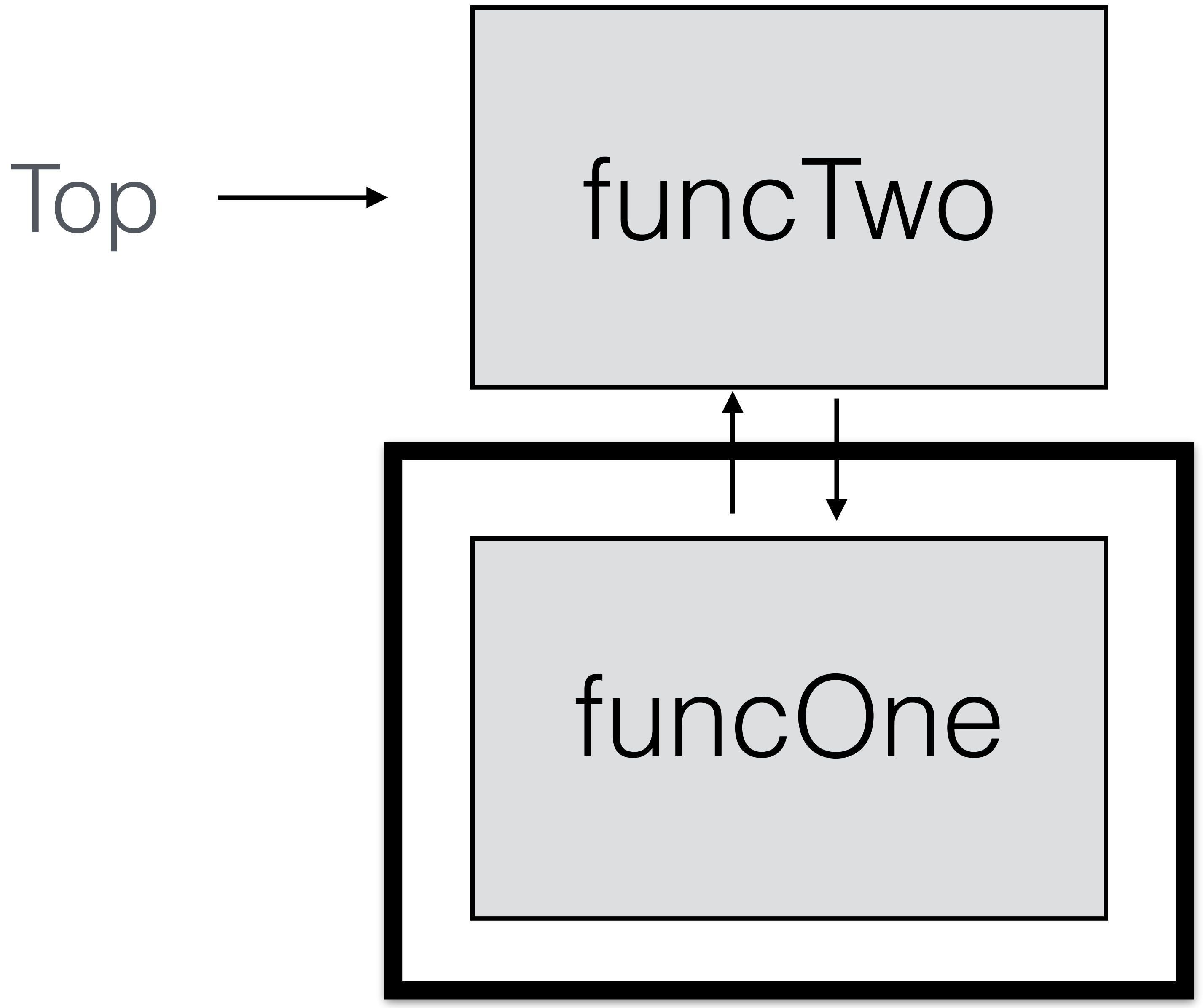
Value 1 returned (stored in next element) and then pop the stack

Call stack: simplified picture

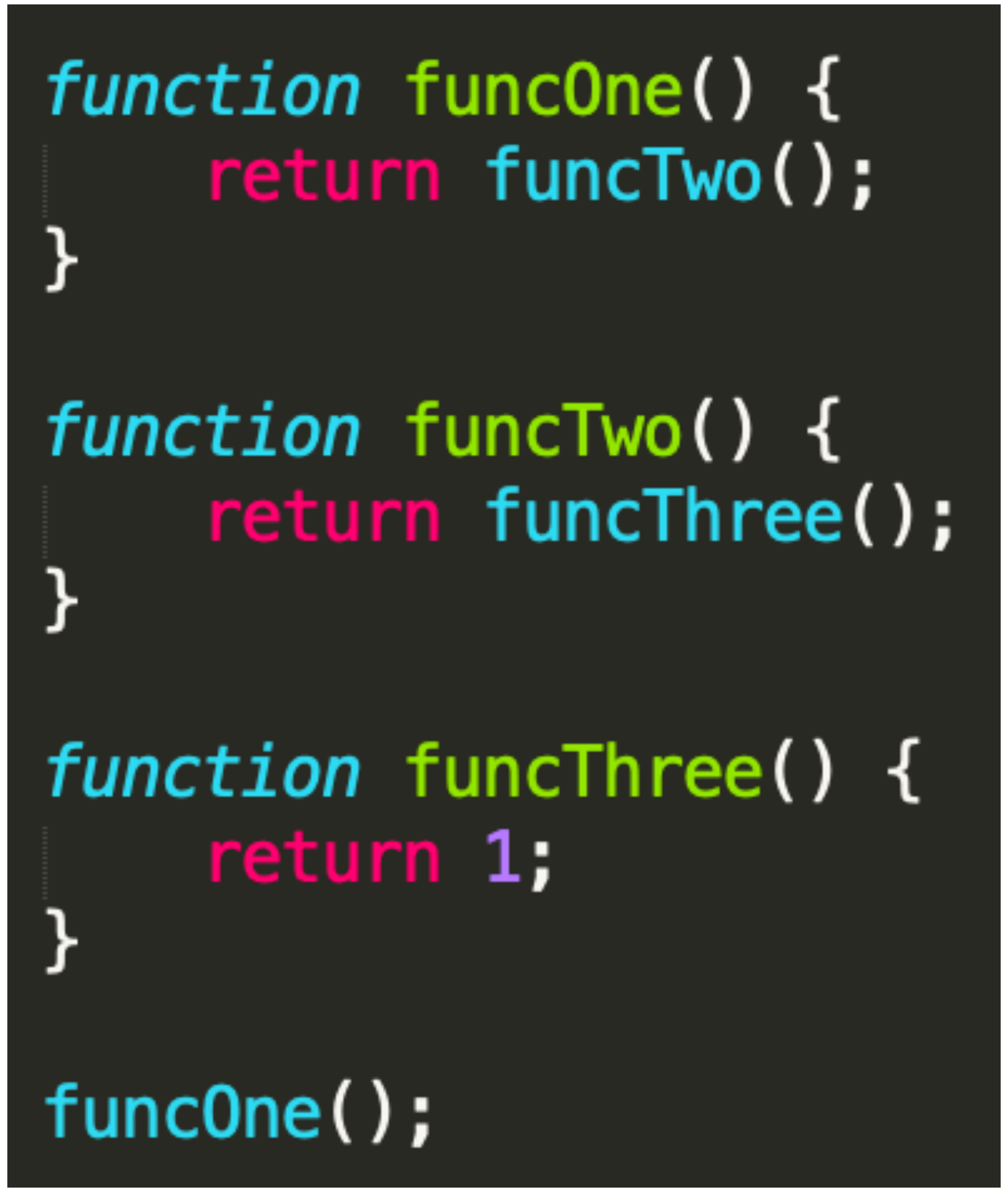


```
function funcOne() {
    return funcTwo();
}

function funcTwo() {
    return funcThree();
}

function funcThree() {
    return 1;
}

funcOne();
```

Returned value (1) passed to top,
pop the stack

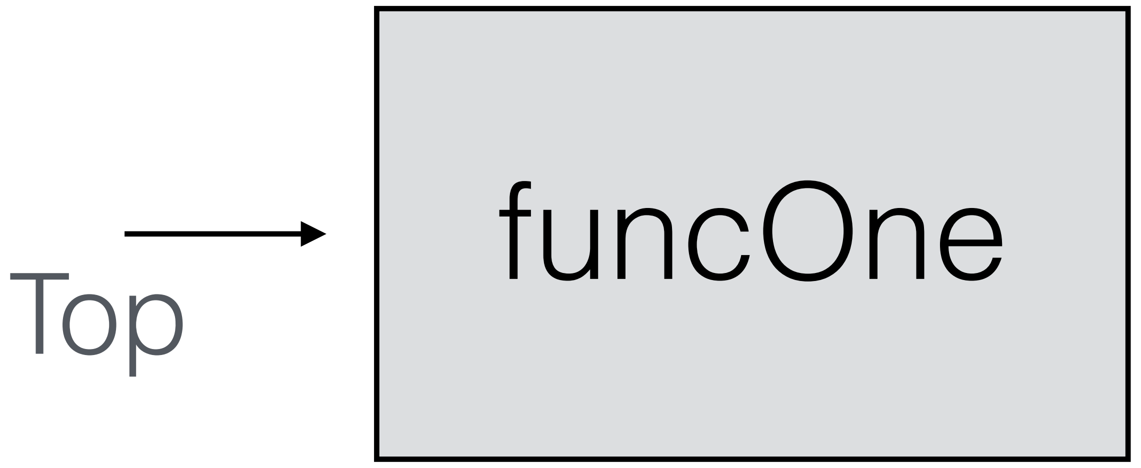


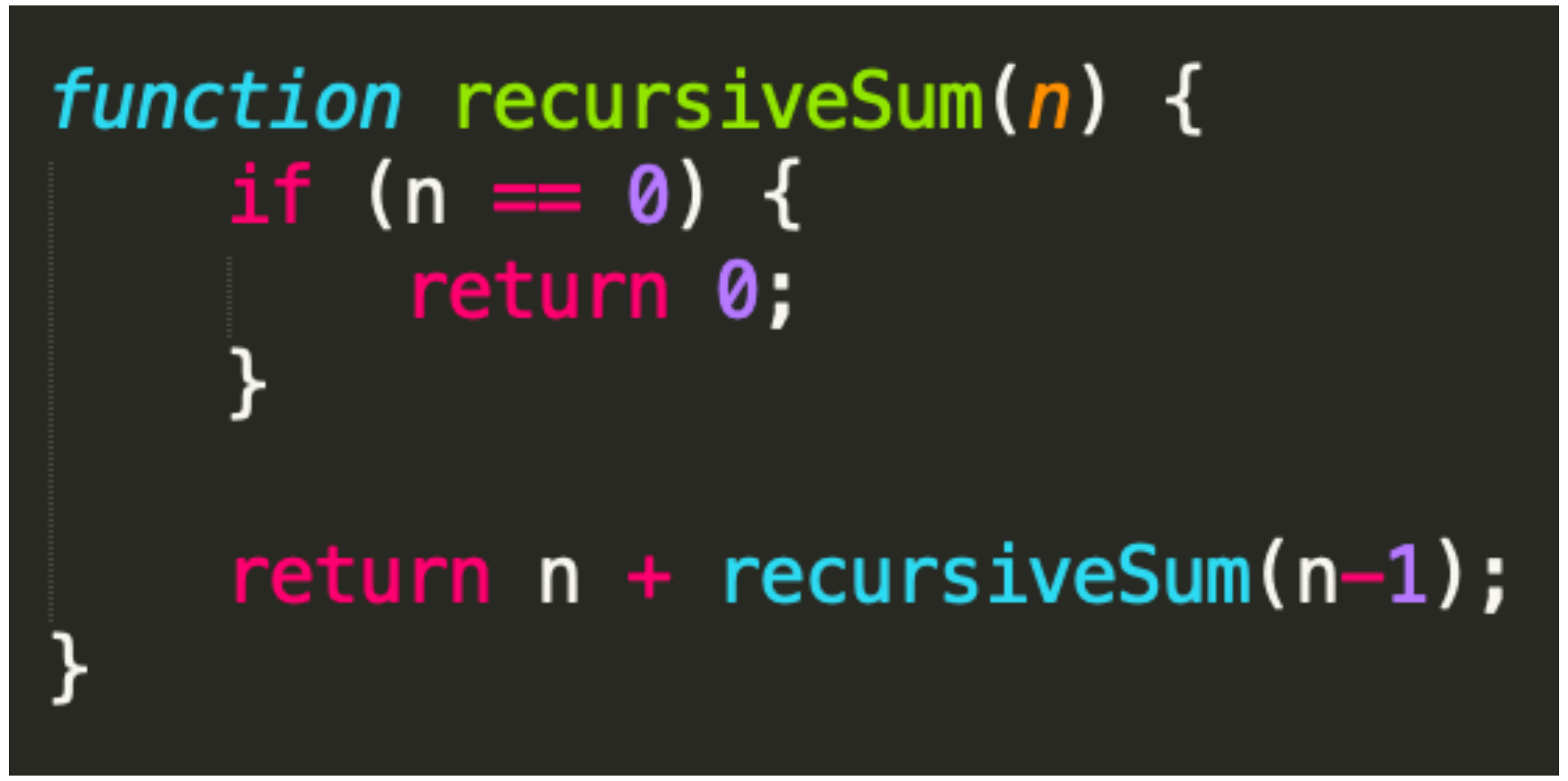
Call stack: simplified picture



```
function funcOne() {
    return funcTwo();
}

function funcTwo() {
    return funcThree();
}

function funcThree() {
    return 1;
}

funcOne();
```

Returned value (1) passed to top,
pop the stack

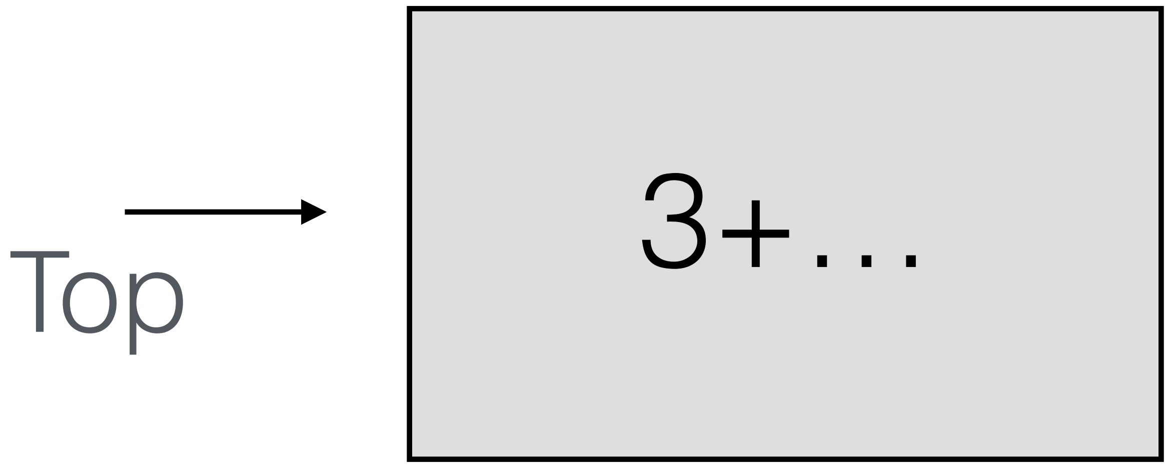




```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n-1);
}
```

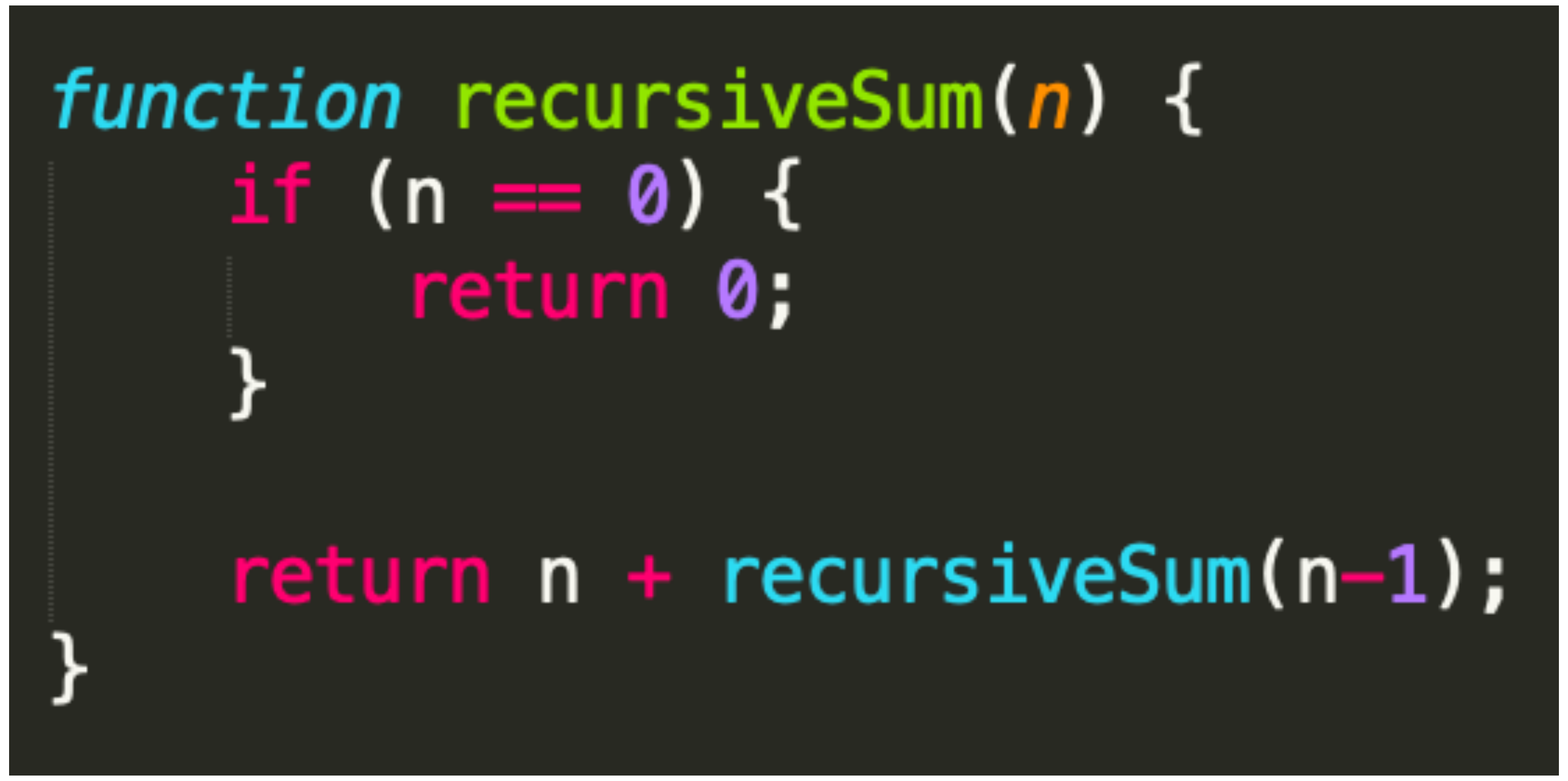
Let’s call recursiveSum(3)

Call recursiveSum(3)



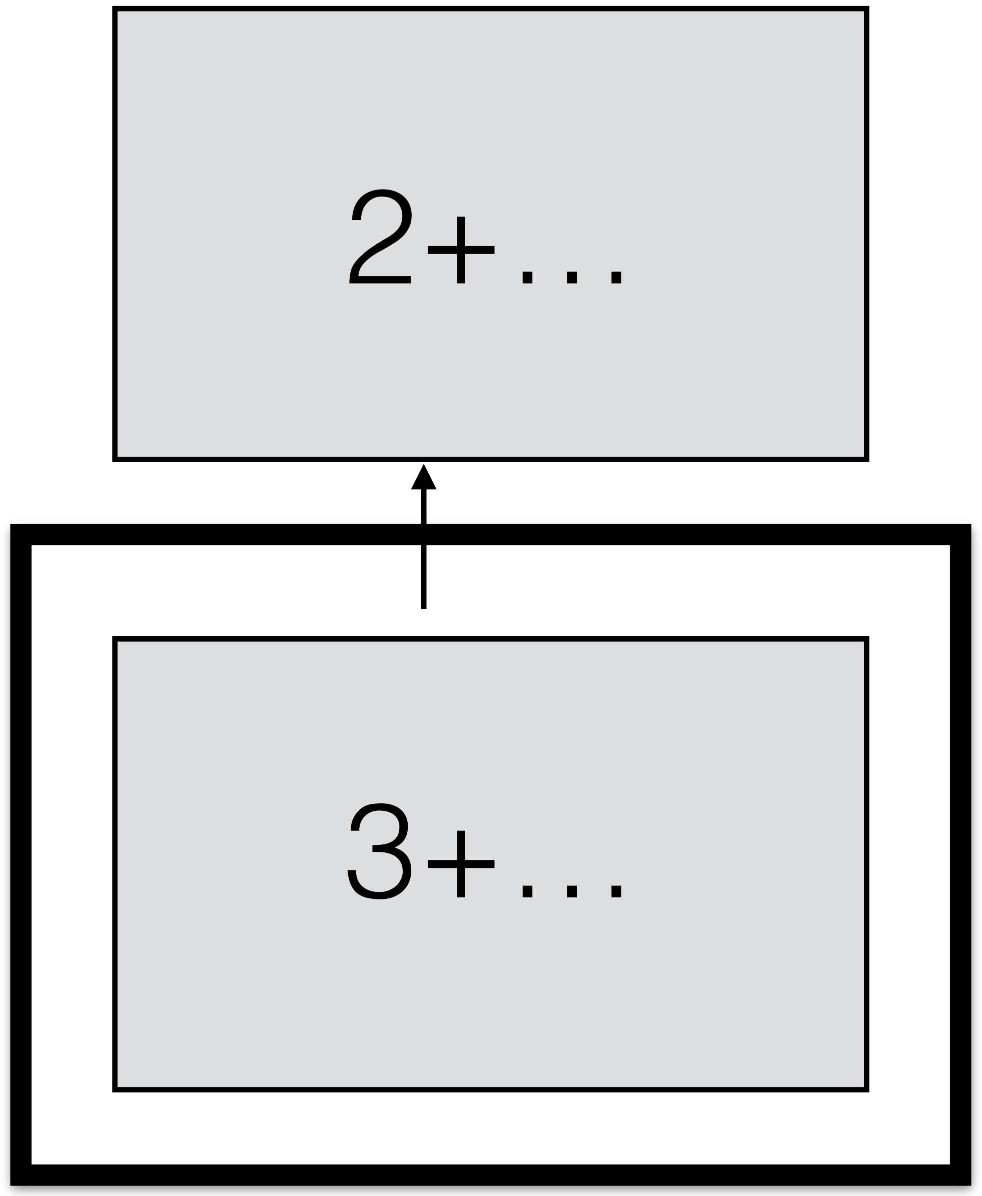
Top

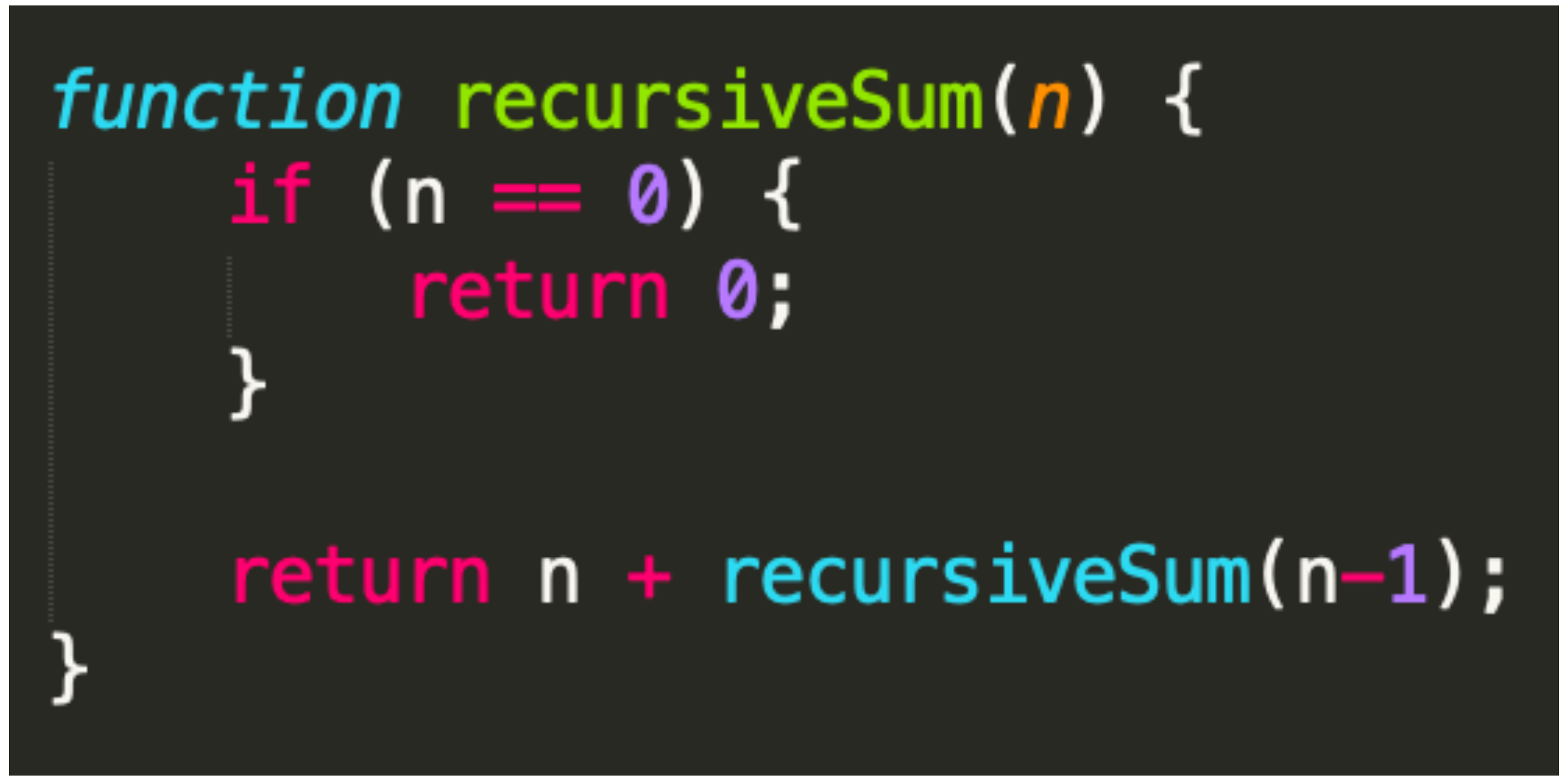
3+…



```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n-1);
}
```

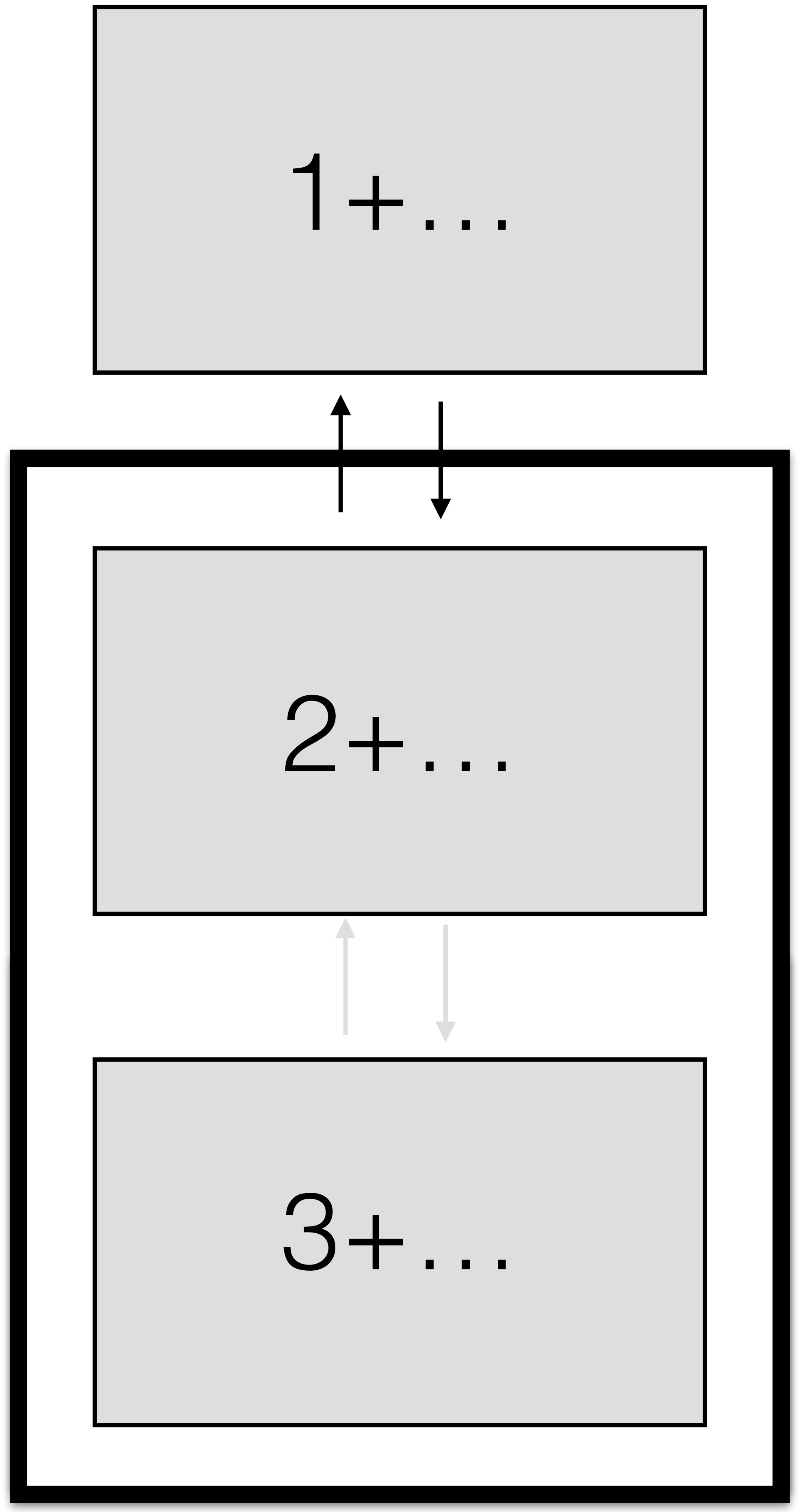
Call recursiveSum(2)

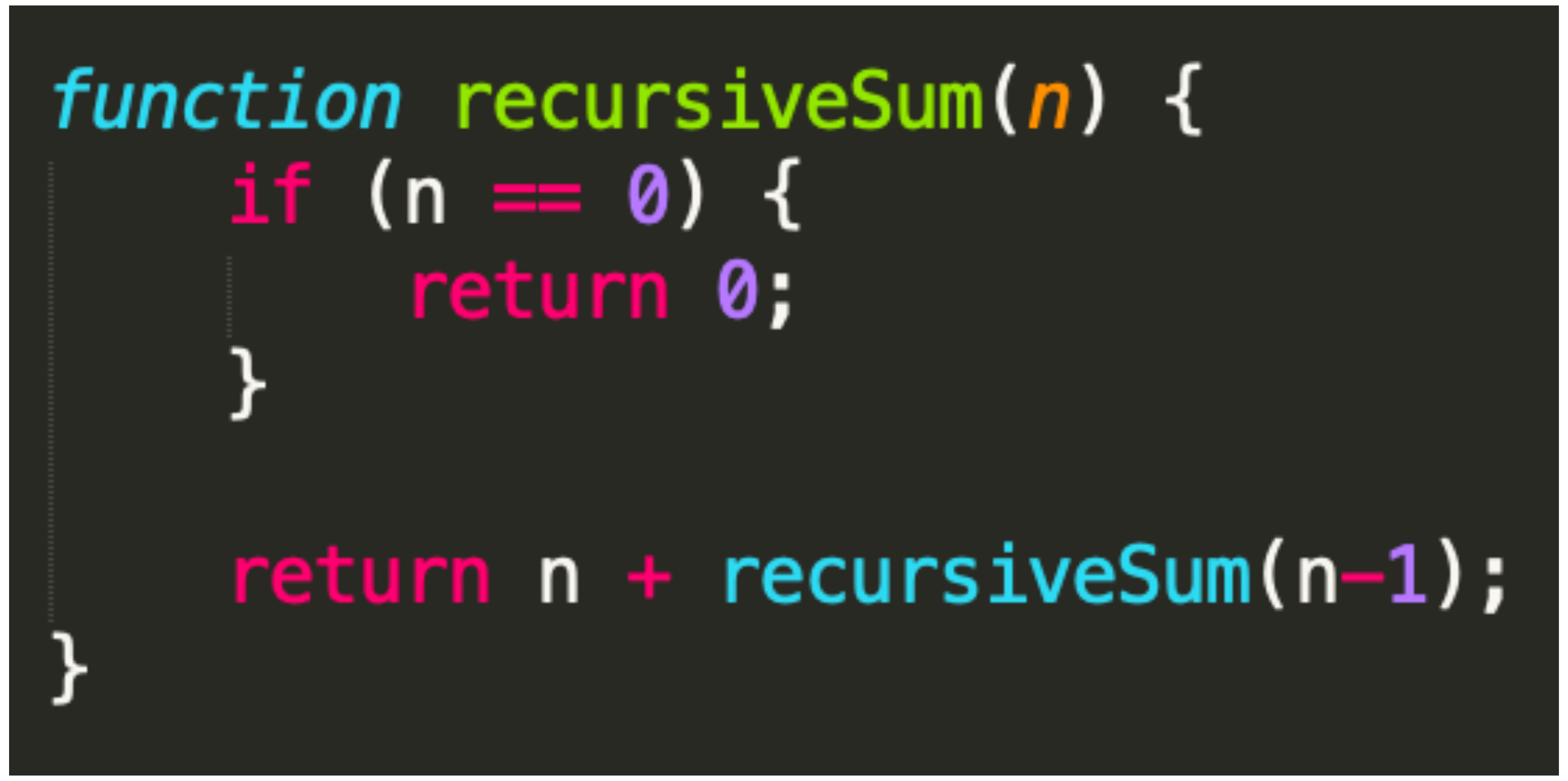




```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n-1);
}
```

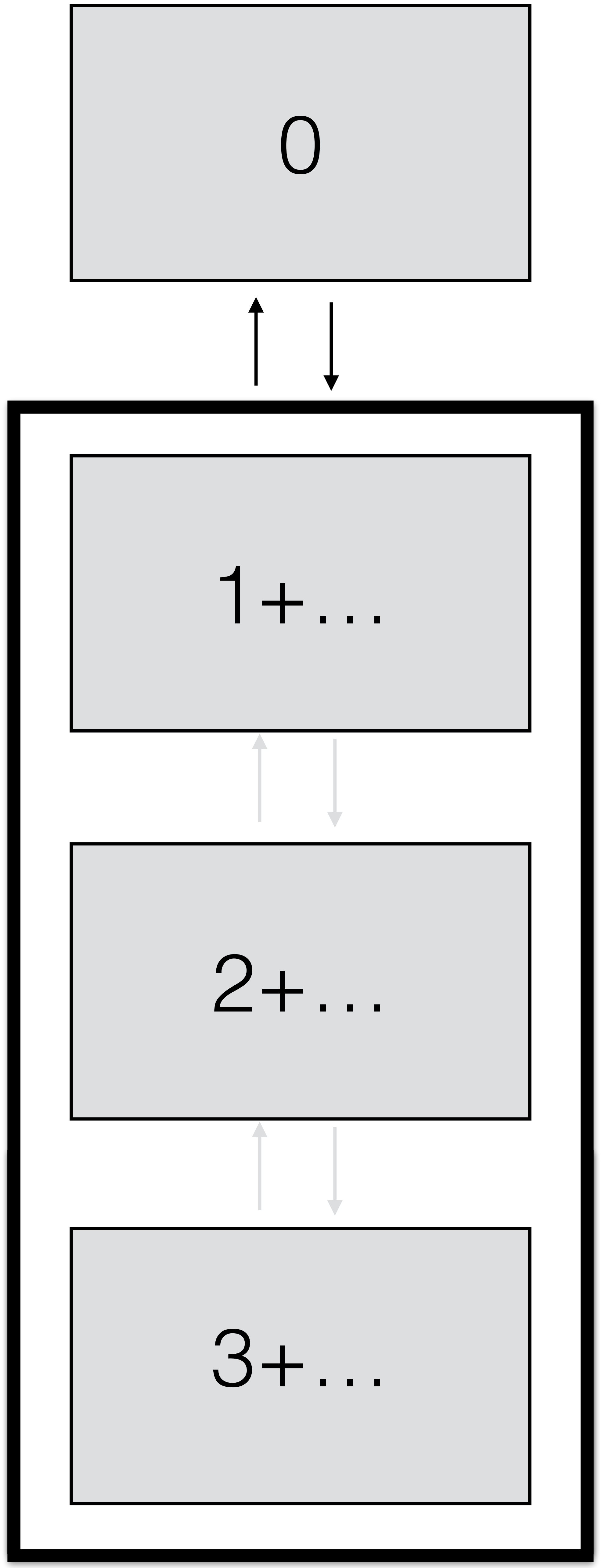
Call recursiveSum(1)

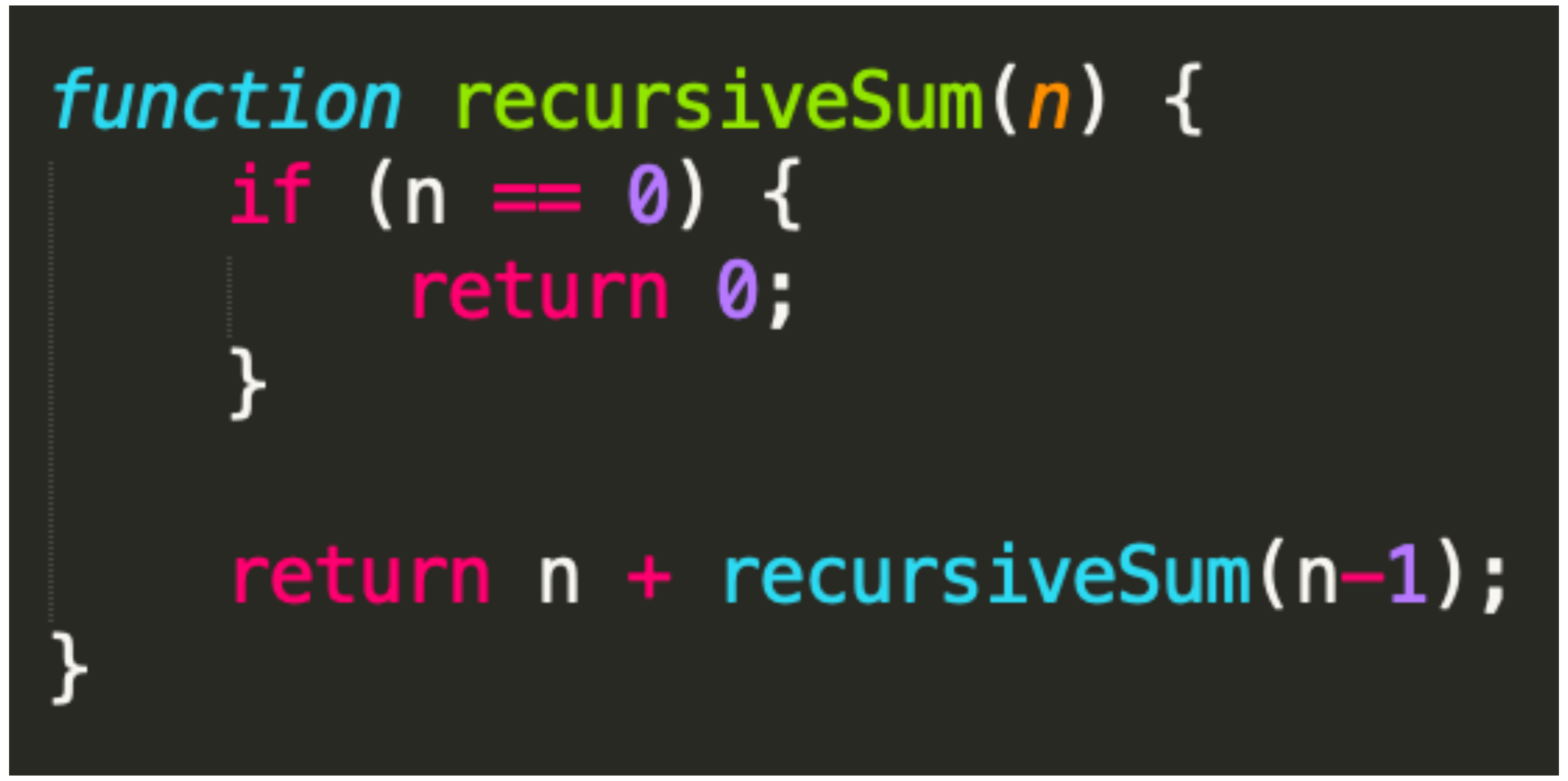




```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n-1);
}
```

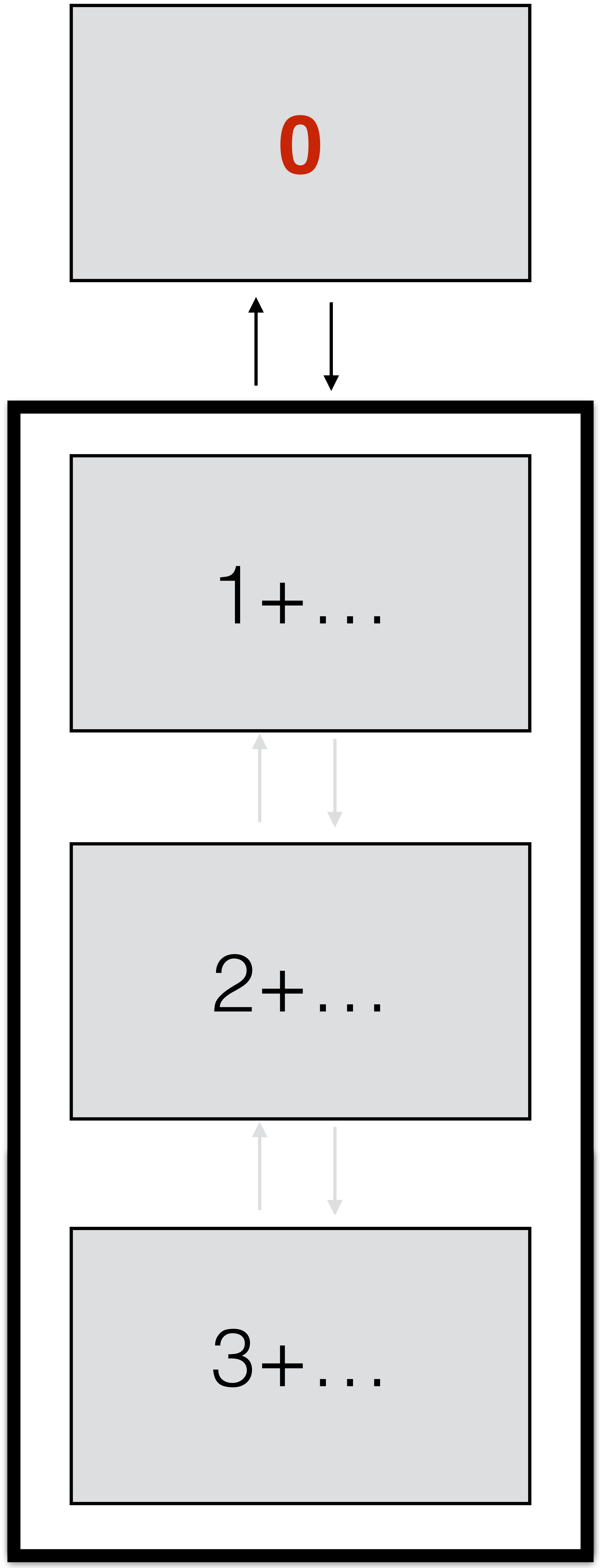
Call recursiveSum(0)

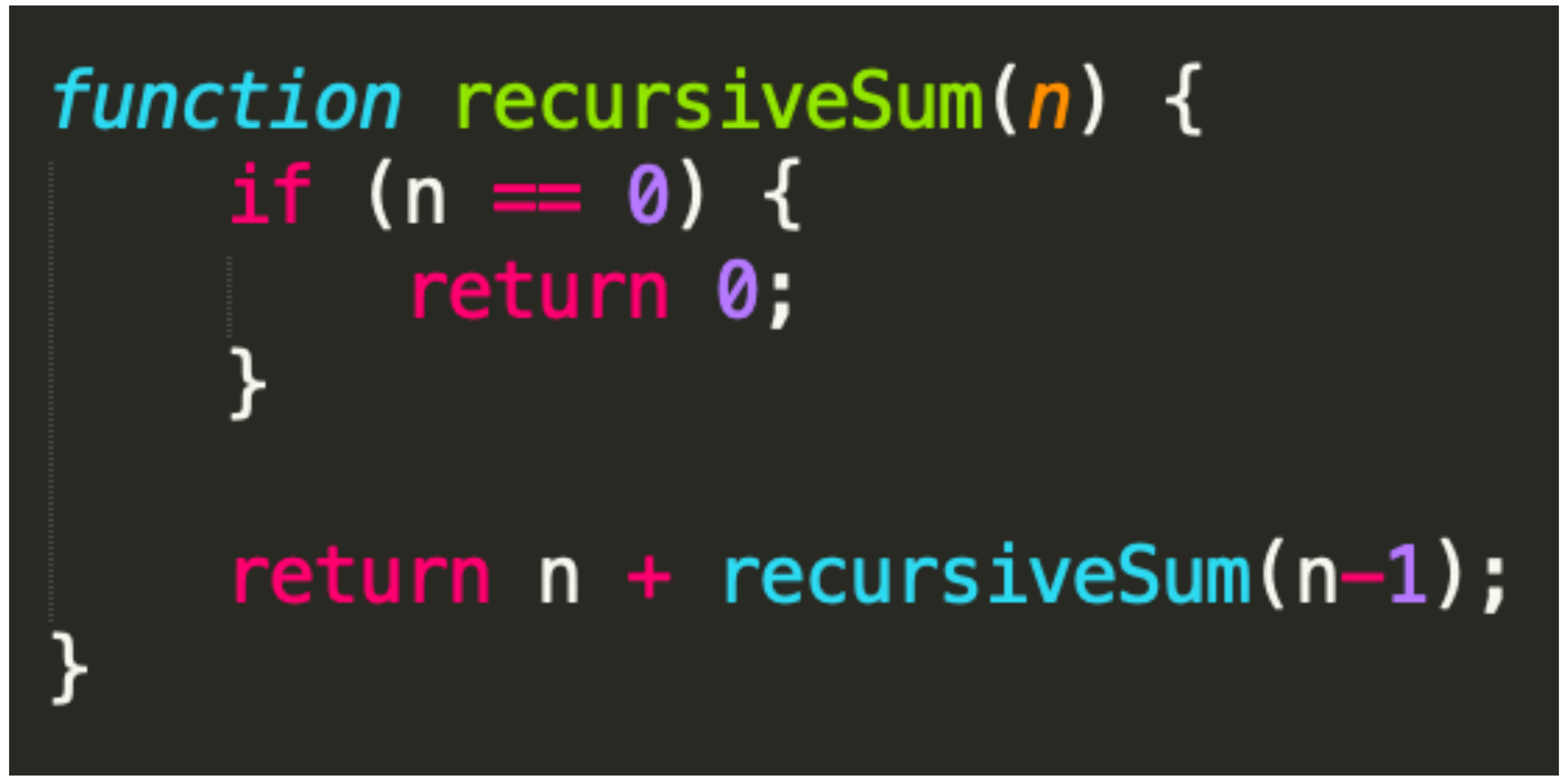




```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n-1);
}
```

Return 0, pop the stack

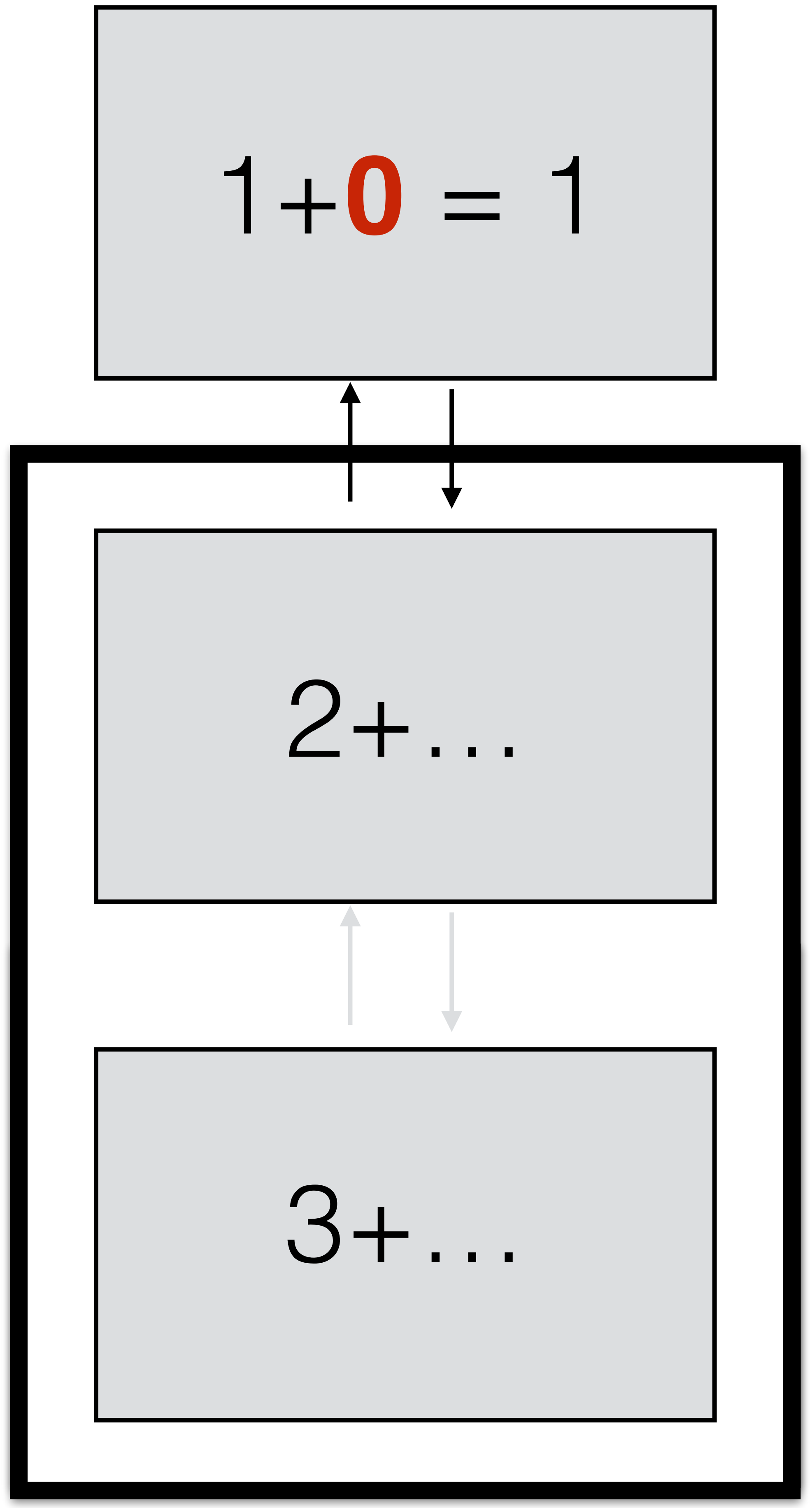




```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n-1);
}
```

Return 0, pop the stack

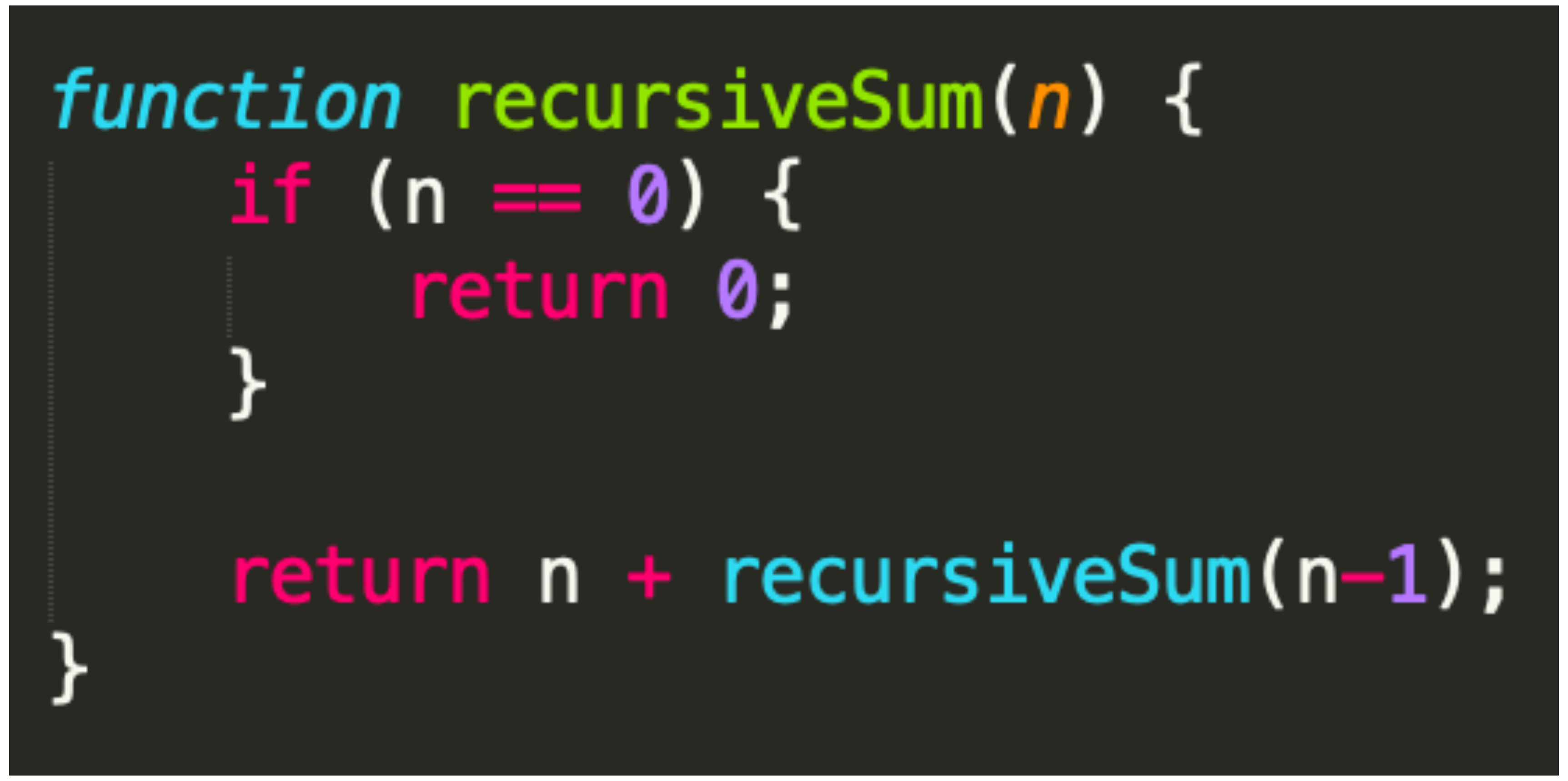
Return 1, pop the stack



1+0 = 1

2+…

3+…

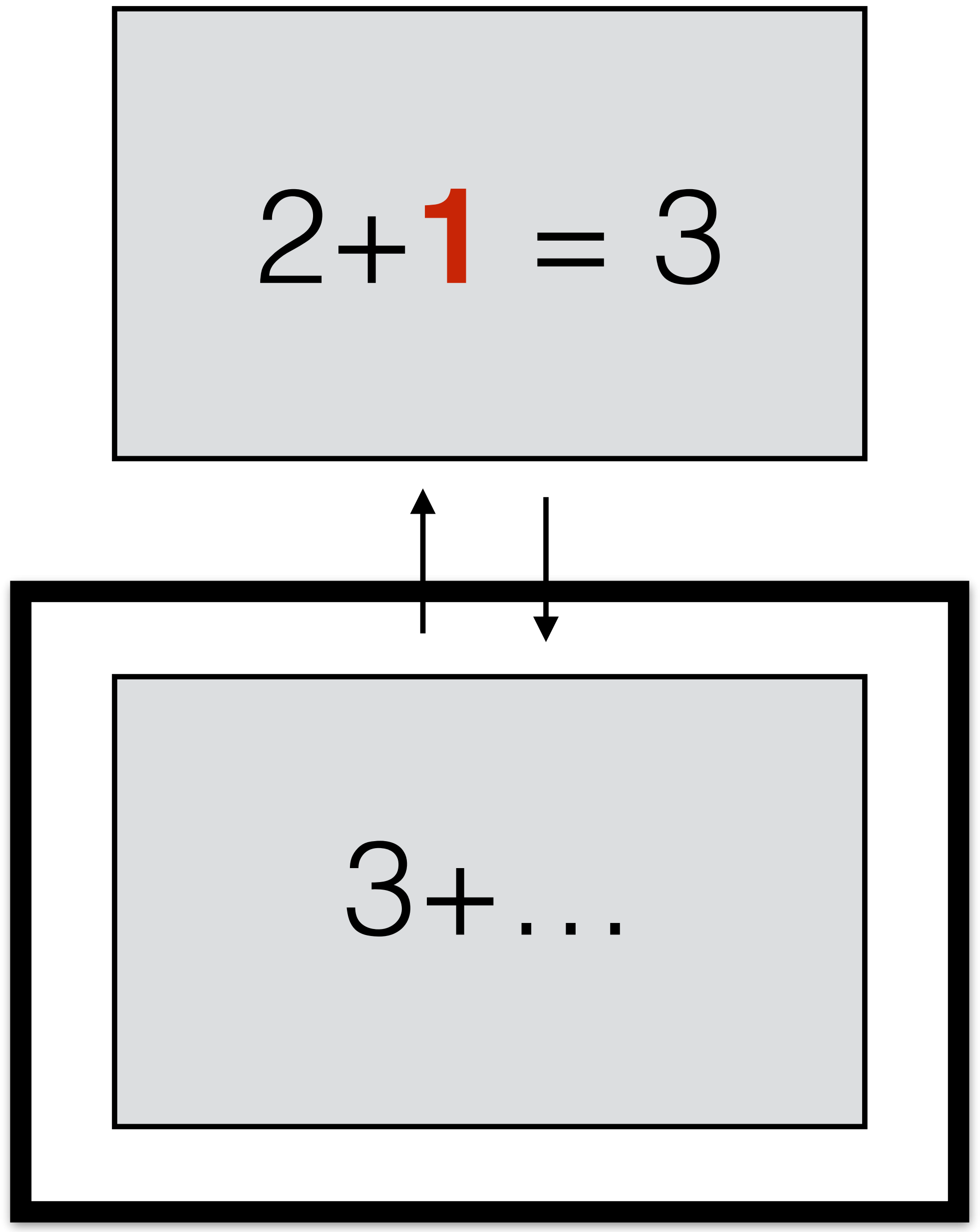


```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n-1);
}
```

Return 0, pop the stack

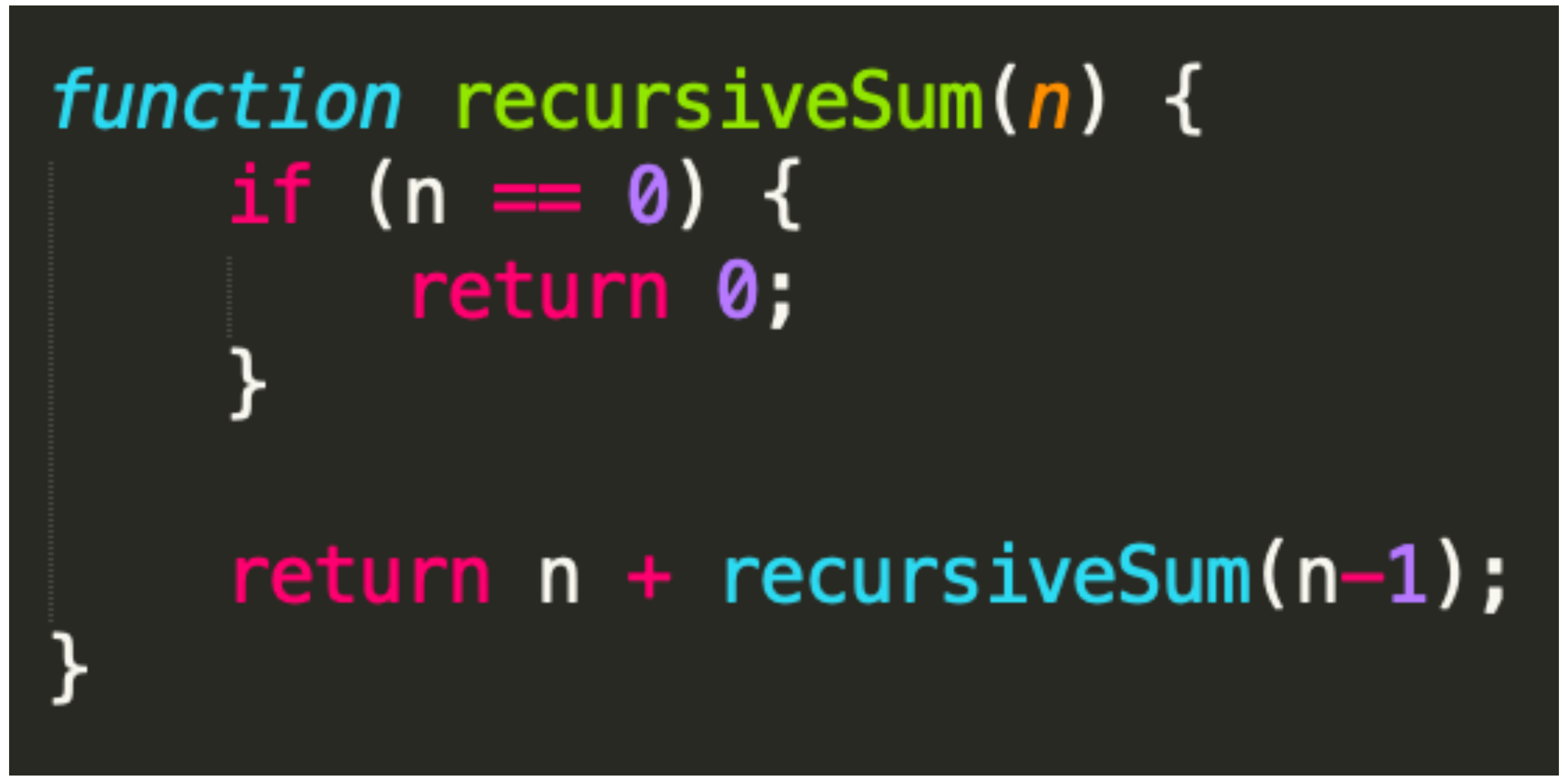
Return 1, pop the stack

Return 3, pop the stack



2+1 = 3

3+…



```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n-1);
}
```

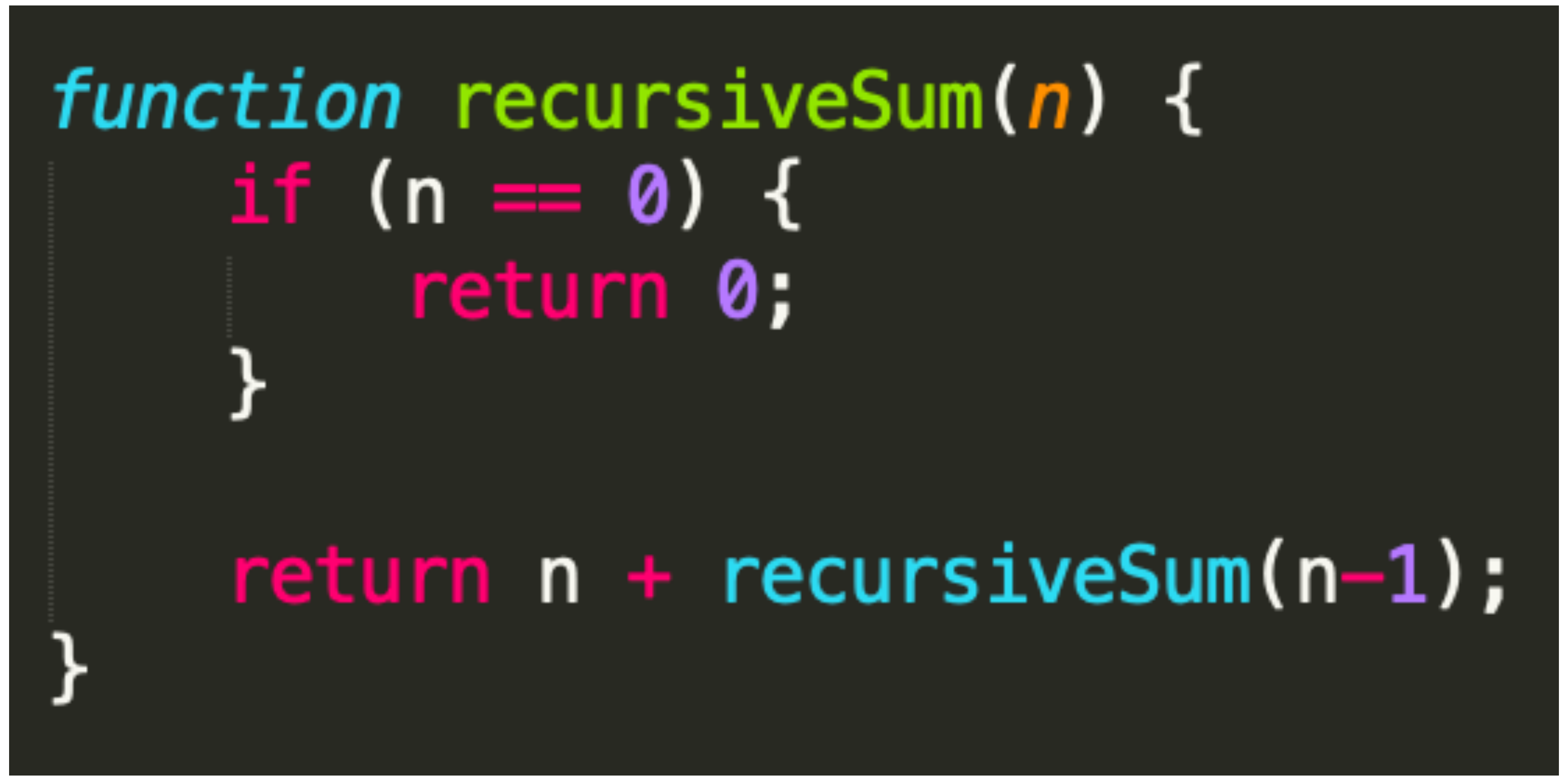
Return 0, pop the stack

Return 1, pop the stack

Return 3, pop the stack

Return 6, empty the stack

3+**3** = 6



```
function recursiveSum(n) {
    if (n == 0) {
        return 0;
    }

    return n + recursiveSum(n-1);
}
```

Return 0, pop the stack

Return 1, pop the stack

Return 3, pop the stack

Return 6, empty the stack

For Review Seminar

In *stack.js* of folder *recursion*

“Simulate” the call stack actions for *recSum* (using iteration or otherwise) in *stackSum*

Given this connection between recursion and stacks

We can turn solution methods involving stacks into ones just using recursion

The stack is now “implicit”

Problem 6:

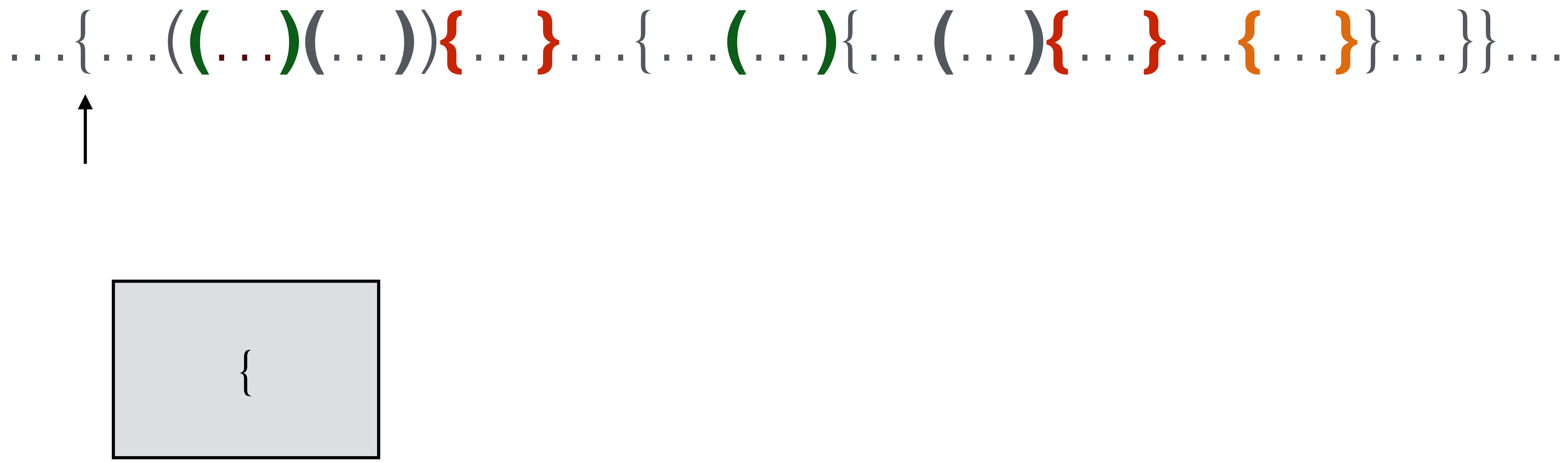
You’ve been given the task of helping to build a “pre-compiler” for a JavaScript teaching tool

This will conduct preliminary checks on code to look for syntax errors to avoid compile errors

Your part in this project is to write code that checks for bracketing errors, both {} and ()

Describe an algorithm and/or JavaScript implementation that flags an error when there is a bracketing error in the code

Problem 6:


...{...((...)(...)){...}...{...(...){...(...){...}...{...}}...}}...

{

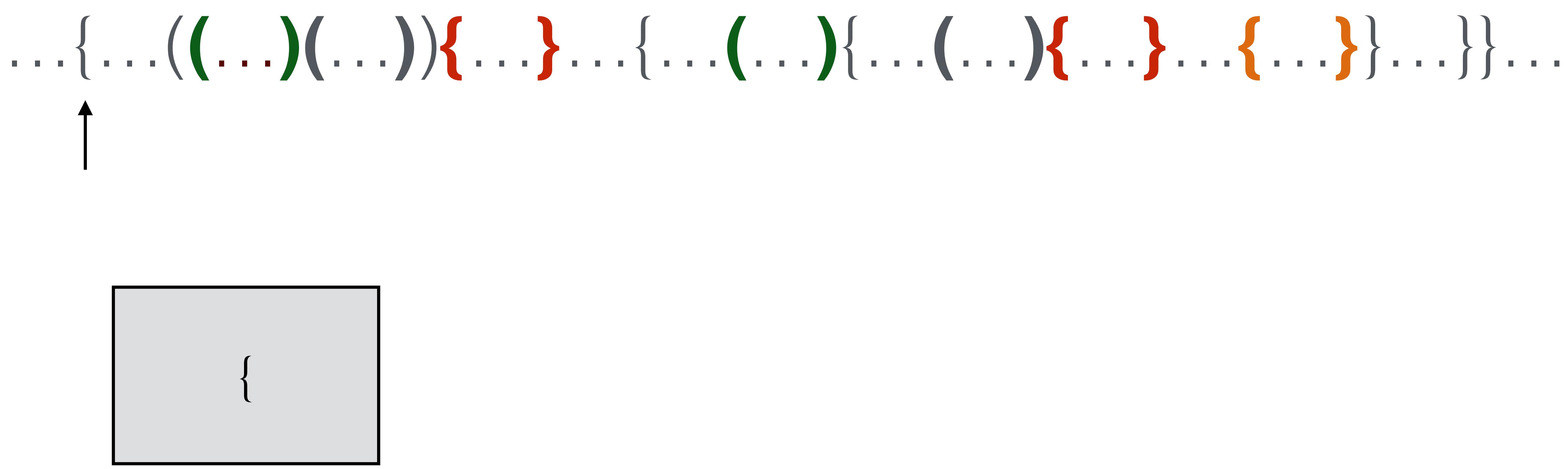
Instead of pushing this to a stack, make a function call

This has the same result of pushing data to a stack

We will pass the number of open brackets to the function and the rest of the string

When we encounter a closed bracket decrease the number of open brackets

Problem 6:



…{…((…)(…)){…}…{…(…){…(…){…}…{…}}…}}…

{

In *recursion.js* let’s look at the function *checkBrackets*

In your own time, complete the functions *findNext* and *checkBrackets* to check non-curly brackets as well

We will go through solution in Review Seminar

Making Sorting Algorithms Recursive

Think about how the Insertion Sort and Bubble Sort algorithms can be made recursive

Try and implement them in the file *sort.js* in folder *recursion*

For Review Seminar

Problem 7:

In Worksheet 3 you learnt about the Ceasar cipher

Letters in the alphabet are *cyclically* permuted

A more secure encryption scheme is to apply *any* permutation of the alphabet (26! many)

abcd

Can you find a ***recursive*** method to put all possible permutations of these four letters in an array?