UMassAmherst | Manning College of Information & Computer Sciences

Programming Methodology

Lab 5

Wednesday, March 6th, 2024

Weekly Lab Agenda

- Go over reminders/goals
- Review past material
- Work in groups of 2-3 to solve a few exercises
    - Please sit with your group from last week.
- Discussion leaders will walk around and answer questions
- Solutions to exercises will be reviewed as a class
- Attendance taken at the end

Reminders

- Midterm 1 is tonight
    - Good luck everyone! :)
- Office hours will end at 6pm tonight.

Today’s Goals

- Midterm Review

Exam Logistics + Advice

Location

  a. Herter 231 – 9:05am and 1:25pm labs
  b. Bartlett 65 – 11:15pm and 12:20pm labs

1. Do not prioritize doing the exam in order!
   b. The order of the questions means nothing. Move on to the next question if you are stuck.
2. Do not leave any question blank!
   b. Don’t write garbage or throw away comments. Write down notes of what you know, that might jog your memory.
3. Write notes to yourself or underline important statements!
   b. What does this function take in? What it returns? What is being graded?
4. Ask questions!
   b. If something doesn’t make sense, raise your hand and a proctor will come over to help out.

Type Signatures

Type Signatures

What are the types of g, a and x?

```
const g = (a, x) => a.map(f => (x => f(f(x)))).map(f => f(x));
```

- Look at the smallest atomic parts of this function.
- Write down what you know immediately.
- Deduce the rest using a combination of what you figured out previously.

Type Signatures

```
const g = (a, x) => a.map(f => (x => f(f(x)))).map(f => f(x));
```

- *a* is an Array.
    - For now we will say its a generic array, we might later figure out what it is *a: T[]*
- ‘*.map*’ iterates over all elements of *a*
    - The *f*’s in this expression ‘*.map(f => (x => f(f(x))))*’ are the elements of *a*
    - They are being called -> they are functions
        - They take in the same type they return
    - ***a: ((x: T) => T)[]   x: T***
- ‘*a.map(f => (x => f(f(x))))*’ is an array of functions of type ‘*(x: T) => T*’
    - ‘*.map(f => f(x))*‘ must return an array, calling the functions made in the previous expression will result in T
- Answer: **g<T>(a: ((x: T) => T)[], x: T): T[]**

Higher Order Functions

Higher Order Functions (HOFs)

```
function reducer(acc, e) {
    return {
        x: acc.y + e,
        y: acc.x
    };
}
const pair = a => a.reduce(reducer, {x: "", y: ""});
```

What is **pair(["that", "is", "a", "short", "text"])** ?

```
function reducer(acc, e) {
   return {
      x: acc.y + e,
      y: acc.x
   };
}
```

reducer switches the value of x and y AND concatenates the current element e onto the value of x

index of array	["that", "is", "a", "short", "text"] initialValue {x:"", y:""}
0th	acc = {x:"", y:""} e = "that" ⟶ {x: "that", y:""}
1st	acc = {x: "that", y:""} e = "is" ⟶ {x: "is", y: "that"}
2nd	acc = {x: "is", y: "that"} e = "a" ⟶ {x: "that a", y: "is"}
3rd	acc = {x: "that a", y: "is"} e = "short" ⟶ {x: "is short", y: "that a"}
4th	acc = {x: "is short", y: "that a"} e = "text"

{x: "that a text", y: "is short"}

Mental Models + Closures

Mental Models

For the line defining o1 below, state how many objects except { } are created when executing that line. State what values are printed. Explain your answers.

```
function f1(k) {
  let o = {};
  while (--k >= 0) {
    o = {val: () => k, next: o}
  }
  return o;
}

let o1 = f1(3);
console.log(o1.val());
console.log(o1.next.val());
```

Mental Models

For the line defining o1 below, state how many objects except { } are created when executing that line. State what values are printed. Explain your answers.

```
1.   function f1(k) {
2.     let o = {};
3.     while (--k >= 0) {
4.       o = {val: () => k, next: o}
5.     }
       return o;
     }

1.   let o1 = f1(3);
2.   console.log(o1.val());
3.   console.log(o1.next.val());
```

The loop at line 3 executes three times; three objects are created and linked, next references the previous created object. The three val closures share the same environment and refer to the same variable k, they are identical.

Calling the closure val evaluates k; by this time, f1 has completed and k is -1, this value is printed.

o1.next.val is an identical closure () => k. This prints -1, the value of k.

Mental Models

When f1 starts executing, k=3 is in the value map.

```
function f1(k) {
  let o = {};
  while (--k >= 0) {
    o = {val: () => k, next: o}
  }
  return o;
}

let o1 = f1(3);
console.log(o1.val());
console.log(o1.next.val());
```

Value

k=3

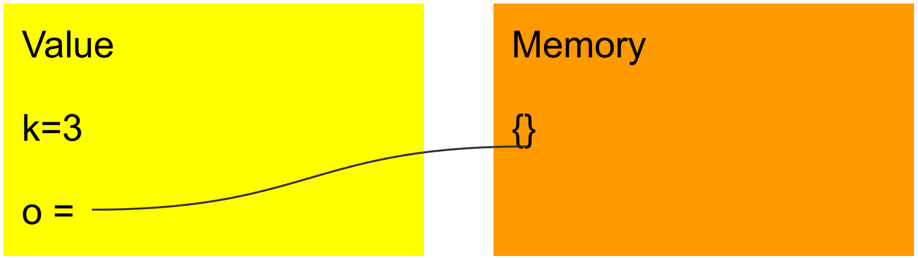
Memory

Mental Models

Line 2 creates an empty object is created in memory and stores its reference in o.

```
1.   function f1(k) {
2.     let o = {};
3.     while (--k >= 0) {
4.       o = {val: () => k, next: o}
5.     }
       return o;
     }

1.   let o1 = f1(3);
2.   console.log(o1.val());
3.   console.log(o1.next.val());
```

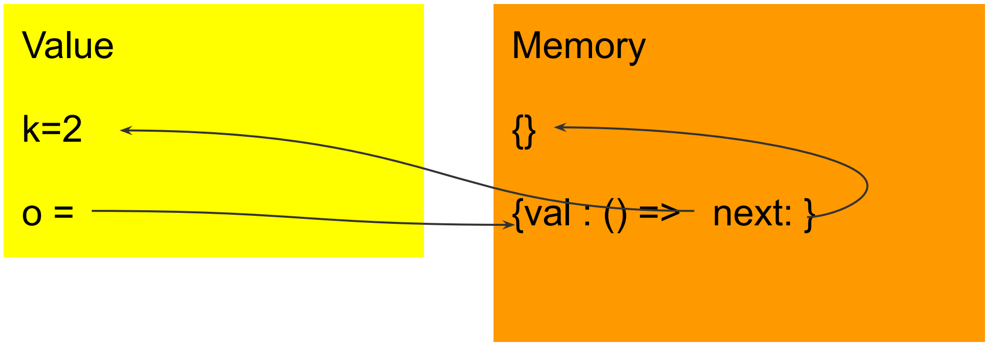


Mental Models

First iteration of the loop decreases value of k by 1 and creates a new object, next field of the newly created object copies reference in stored in o and o stores reference to the newly created object.

```
function f1(k) {
  let o = {};
  while (--k >= 0) {
    o = {val: () => k, next: o}
  }
  return o;
}

let o1 = f1(3);
console.log(o1.val());
console.log(o1.next.val());
```

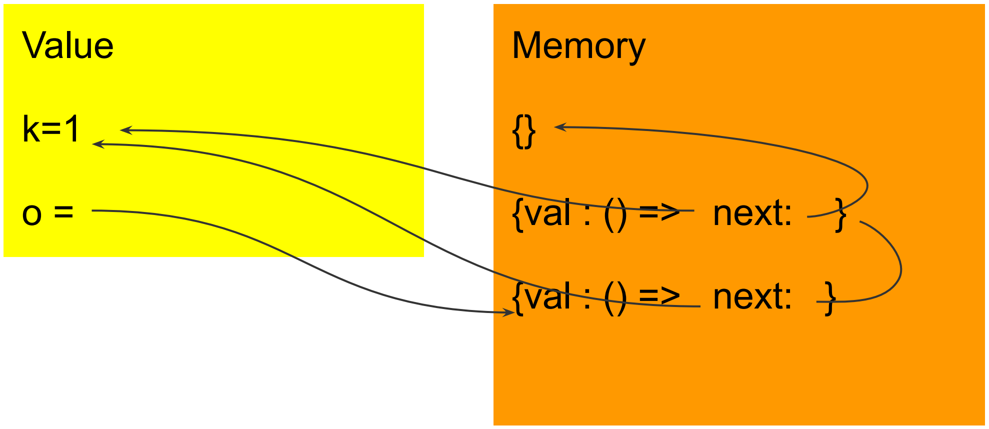


Mental Models

Second iteration of the loop another object is created

```
function f1(k) {
  let o = {};
  while (--k >= 0) {
    o = {val: () => k, next: o}
  }
  return o;
}

let o1 = f1(3);
console.log(o1.val());
console.log(o1.next.val());
```

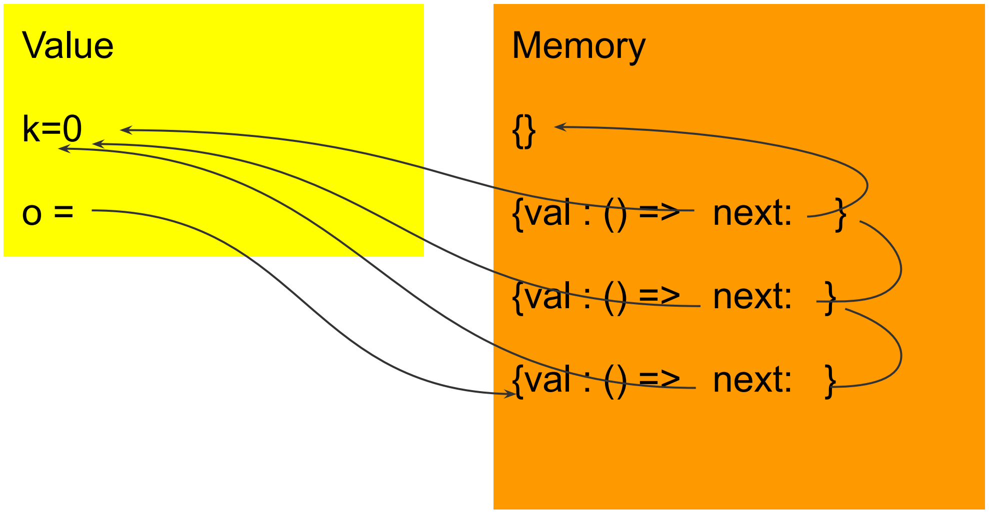


Mental Models

Third iteration of the loop. One more object created

```
function f1(k) {
  let o = {};
  while (--k >= 0) {
    o = {val: () => k, next: o}
  }
  return o;
}

let o1 = f1(3);
console.log(o1.val());
console.log(o1.next.val());
```

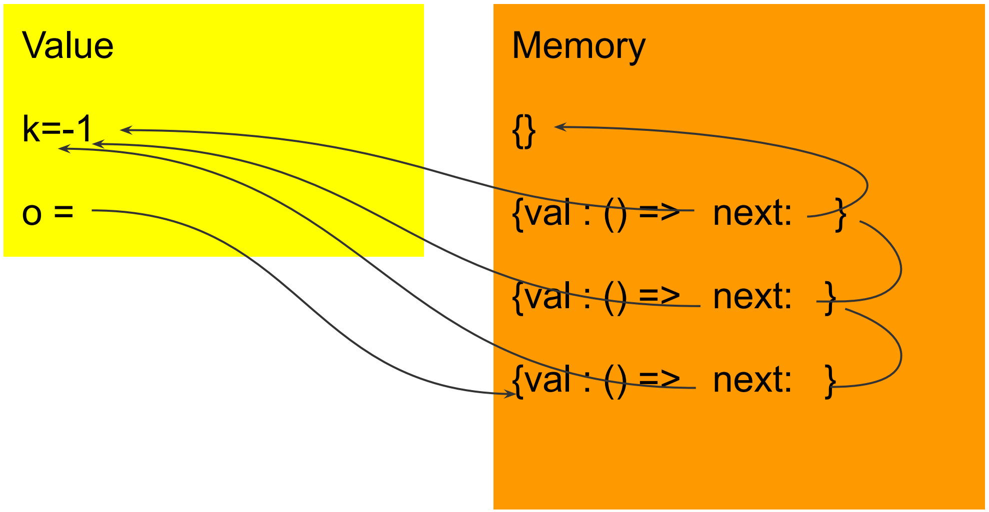


Mental Models

While condition is evaluated, which decreases k by 1. Body of the loop isn't executed.

```
function f1(k) {
  let o = {};
  while (--k >= 0) {
    o = {val: () => k, next: o}
  }
  return o;
}

let o1 = f1(3);
console.log(o1.val());
console.log(o1.next.val());
```

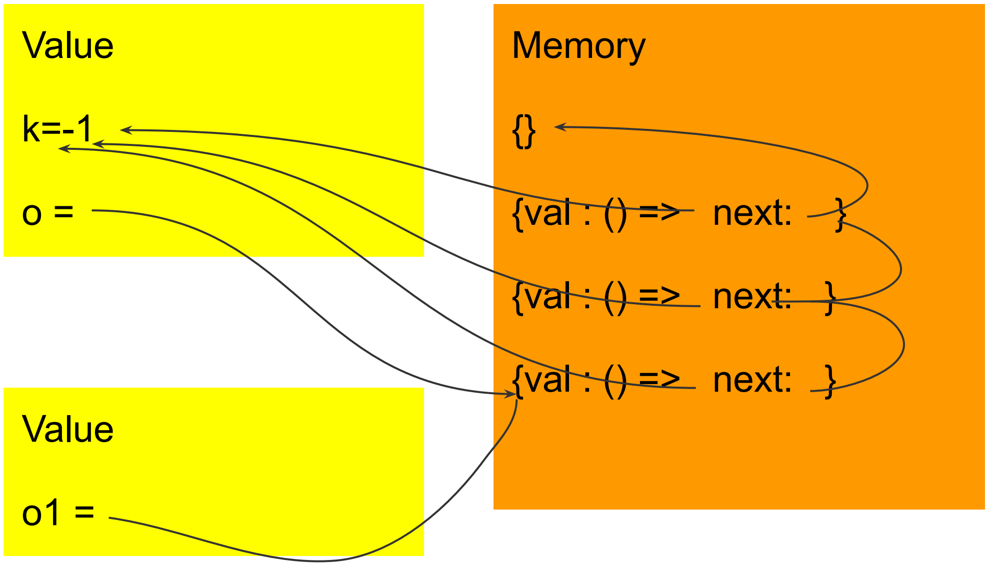


Mental Models

Upon return, o1 will have the reference value returned through o.

```
function f1(k) {
  let o = {};
  while (--k >= 0) {
    o = {val: () => k, next: o}
  }
  return o;
}

let o1 = f1(3);
console.log(o1.val());
console.log(o1.next.val());
```



Lists

Fall 2020 Midterm 1: Interleaving Lists

Write a function that takes two lists of type `List<T>` and interleaves them. The result should be a new list which alternates elements from the two lists, starting with the first element of the first list, and stopping when no element can be extracted from a list. For full credit, write a single recursive function with a single decision or conditional.

Hint: Think back to the List merge example we saw in lab a couple weeks ago.

Fall 2020 Midterm 1: Interleaving Lists

Write a function that takes two lists of type `List<T>` and interleaves them. The result should be a new list which alternates elements from the two lists, starting with the first element of the first list, and stopping when no element can be extracted from a list. For full credit, write a single recursive function with a single decision or conditional.

Solution:

```
function inter<T>(lst1: List<T>, lst2: List<T>): List<T> {
    return lst1.isEmpty()
        ? empty<T>()
        : node(lst1.head(), inter(lst2, lst1.tail()));
}
```

Interleaving Lists Example

```
function inter<T>(lst1: List<T>, lst2: List<T>): List<T> {
    return lst1.isEmpty()
        ? empty<T>()
        : node(lst1.head(), inter(lst2, lst1.tail()));
}
```

```
Original lst1: 1 -> 2 -> empty<T>()
Original lst2: 3 -> 4 -> 5 -> empty<T>()
```

Interleaving Lists Example

```
function inter<T>(lst1: List<T>, lst2: List<T>): List<T> {
    return lst1.isEmpty()
        ? empty<T>()
        : node(lst1.head(), inter(lst2, lst1.tail()));
}
```

lst1: 1 -> 2 -> empty<T>()

lst2: 3 -> 4 -> 5 -> empty<T>()

Returned list: 1

Original lst1: 1 -> 2 -> empty<T>()

Original lst2: 3 -> 4 -> 5 -> empty<T>()

Interleaving Lists Example

```
function inter<T>(lst1: List<T>, lst2: List<T>): List<T> {
    return lst1.isEmpty()
        ? empty<T>()
        : node(lst1.head(), inter(lst2, lst1.tail()));
}
```

lst1: 3 -> 4 -> 5 -> empty<T>()

lst2: 2 -> empty<T>()

Returned list: 1 -> 3

Original lst1: 1 -> 2 -> empty<T>()

Original lst2: 3 -> 4 -> 5 -> empty<T>()

Interleaving Lists Example

```
function inter<T>(lst1: List<T>, lst2: List<T>): List<T> {
    return lst1.isEmpty()
        ? empty<T>()
        : node(lst1.head(), inter(lst2, lst1.tail()));
}
```

```
lst1: 2 -> empty<T>()
lst2: 4 -> 5 -> empty<T>()

Returned list: 1 -> 3 -> 2
```

```
Original lst1: 1 -> 2 -> empty<T>()
Original lst2: 3 -> 4 -> 5 -> empty<T>()
```

Interleaving Lists Example

```
function inter<T>(lst1: List<T>, lst2: List<T>): List<T> {
    return lst1.isEmpty()
        ? empty<T>()
        : node(lst1.head(), inter(lst2, lst1.tail()));
}
```

```
lst1: 4 -> 5 -> empty<T>()
lst2: empty<T>()

Returned list: 1 -> 3 -> 2 -> 4
```

```
Original lst1: 1 -> 2 -> empty<T>()
Original lst2: 3 -> 4 -> 5 -> empty<T>()
```

Interleaving Lists Example

```
function inter<T>(lst1: List<T>, lst2: List<T>): List<T> {
    return lst1.isEmpty()
        ? empty<T>()
        : node(lst1.head(), inter(lst2, lst1.tail()));
}
```

```
lst1: empty<T>()
lst2: 5 -> empty<T>()

Returned list: 1 -> 3 -> 2 -> 4 -> empty<T>()
```

```
Original lst1: 1 -> 2 -> empty<T>()
Original lst2: 3 -> 4 -> 5 -> empty<T>()
```

Lists

Consider the following code fragment working with lists as defined in class.
How many list nodes (created with node()) are no longer accessible at the end of this code fragment?

```
let lst1 = ... // create a list with 2 elements
const concat =
  (l1, l2) => l1.isEmpty()? l2 : node(l1.head(), concat(l1.tail(), l2));
lst1 = concat(concat(lst1, lst1), lst1)
// end of the code fragment
```

Hints:
node constructor creates an object of of type List<T> and returns a reference to it.
Every call to the node constructor or to empty() creates a new object in memory.
At the end of the code, we have one variables in value map: lst1.
Objects that are not accessible through lst1 are no longer accessible.

Lists

Take a List<number> for example. Line 1 has two calls to the node constructor and one call to empty. This creates three objects of type List<number> in memory.

```
let lst1 = node(1, node(2, empty()));
const concat =
  (l1, l2) => l1.isEmpty() ? l2 : node(l1.head(), concat(l1.tail(), l2));
lst1 = concat(concat(lst1, lst1), lst1);
```

Value Map

lst1 =

Memory
{head: () => { throw new Error()}, tail: () => { throw new Error()}}
{head: () => 2 ;   tail: () => }
{head: () => 1 ;   tail: () => }

Lists

Line 2 is a function definition, memory remains the same.

```
let lst1 = node(1, node(2, empty()));
const concat =
  (l1, l2) => l1.isEmpty() ? l2 : node(l1.head(), concat(l1.tail(), l2));
lst1 = concat(concat(lst1, lst1), lst1);
```

Value Map

lst1 =

Memory

{head: () => { throw new Error()}, tail: () => { throw new Error()}}

{head: () => 2 ;   tail: () => }

{head: () => 1 ;   tail: () => }

Lists

Line 3 has two function calls to concat which need to be evaluated to determine the reference that gets assigned to lst1.

```
let lst1 = node(1, node(2, empty()));
const concat =
  (l1, l2) => l1.isEmpty() ?  l2 : node(l1.head(), concat(l1.tail(), l2));
lst1 = concat(concat(lst1, lst1), lst1);
```

Value Map

lst1 =

Memory

{head: () => { throw new Error()}, tail: () => { throw new Error()}}

{head: () => 2 ;   tail: () => }

{head: () => 1 ;   tail: () => }

Lists

Inner call to concat calls the node constructor twice. Two more objects are created, after which l1.isEmpty() evaluates to true. Second object created references the old lst1 through the tail() call.

All objects are accessible through lst1. Note that lst1 hasn't changed.

```
let lst1 = node(1, node(2, empty()));
const concat =
  (l1, l2) => l1.isEmpty() ? l2 : node(l1.head(), concat(l1.tail(), l2));
lst1 = concat(concat(lst1, lst1), lst1);
```

Value Map

lst1 =

Memory

{head: () => { throw new Error()}, tail: () => { throw new Error()}}

{head: () => 2 ;   tail: () => }

{head: () => 1 ;   tail: () => }

{head: () => 2 ;   tail: () => }

{head: () => 1 ;   tail: () => }

copy of lst1

Lists

Outer call to concat creates 4 more objects because the first argument concat(lst1, lst1) is a 4-element list. The tail to this 4 element list is set to the reference stored in lst1.

```
let lst1 = node(1, node(2, empty()));
const concat =
  (l1, l2) => l1.isEmpty() ? l2 : node(l1.head(), concat(l1.tail(), l2));
lst1 = concat(concat(lst1, lst1), lst1);
```

Value Map

lst1 =

Memory

{head: () => { throw new Error()}, tail: () => { throw new Error()}}

{head: () => 2 ; tail: () => }

{head: () => 1 ; tail: () => }

{head: () => 2 ; tail: () => }

{head: () => 1 ; tail: () => }

{head: () => 2 ; tail: () => }

{head: () => 1 ; tail: () => }

{head: () => 2 ; tail: () => }

{head: () => 1 ; tail: () => }

copy of concat(lst1, lst1)

Lists

After right hand side of the assignment is evaluated, lst1 is updated. Two list nodes are no longer accessible.

```
let lst1 = node(1, node(2, empty()));
const concat =
  (l1, l2) => l1.isEmpty() ? l2 : node(l1.head(), concat(l1.tail(), l2));
lst1 = concat(concat(lst1, lst1), lst1);
```

Value Map

lst1 =

Memory

{head: () => { throw new Error()}, tail: () => { throw new Error()}}
{head: () => 2 ; tail: () => }
{head: () => 1 ; tail: () => }
{head: () => 2 ; tail: () => }
{head: () => 1 ; tail: () => }

These two objects aren’t reachable through lst1.

{head: () => 2 ; tail: () => }
{head: () => 1 ; tail: () => }
{head: () => 2 ; tail: () => }
{head: () => 1 ; tail: () => }

Testing

Testing

The findIndex function returns the index of the first element of array a for which f returns true, or -1 if no such element exists

```
// findIndex<T>(a: T[], f: T => boolean): number
function findIndex(a, f) {
    for (let i = 0; i < a.length; ++i) {
        if (f(a[i])) { return i; }
    }
    return -1;
}
```

Write three representative cases that you would use to test findIndex.
You need not write code, but clearly indicate inputs, output, and the purpose of the test

Testing

```
// findIndex<T>(a: T[], f: T => boolean): number
function findIndex(a, f) {
    for (let i = 0; i < a.length; ++i) {
        if (f(a[i])) { return i; }
    }
    return -1;
}
```

Three example tests:

`a: [ ], f: x => x > 0`. Output: -1. No element is found for an empty array.
`a: [4, 3, 2, 5], f: x => x % 2 > 0`. Output: 1. Test first index is returned for multiple matches.
`a: [‘hi’, ‘ho’], f: s => s.length > 2`. Output: -1. Test case when no element matches.

GOOD LUCK!!!

Type Signatures

```
const g = (a, x) => a.map(f => (x => f(f(x)))).map(f => f(x));
```

```
map : ((T => T)[], (T => T) => T) => T[]

map : ((T => T)[], (T => T) => (T => T))
      => (T => T)[]

h => h(x)

: (T => T) => T

f : (T => T)

a : A[]

f => (t: T => f(f(t)))

: A => (T => ?)

: (T => T)[]

: (T => T) => (T => T)
```

More Higher Order Functions

Write a function f: number[][]=>boolean that takes a 2D array of numbers and returns true if and only if every row contains at least one even number. Do not use loops or recursion.
let array = [[1,2,3], [4,5,6], [7,8], [9, 10]] //f(array) → true
let array2 = [[1,3], [4,5,6], [7,8], [9]] //f(array2) → false

More Higher Order Functions

Write a function f: number[][]=>boolean that takes a 2D array of numbers and returns true if and only if every row contains at least one even number. Do not use loops or recursion.
let array = [[1,2,3], [4,5,6], [7,8], [9, 10]] //f(array) → true
let array2 = [[1,3], [4,5,6], [7,8], [9,]] //f(array2) → false
const f = (a) => a.every((row) => row.some((num) => num % 2 === 0))
const f2 = (a) => a.reduce((acc, e) => acc && e.some((num) => num % 2 === 0), true)