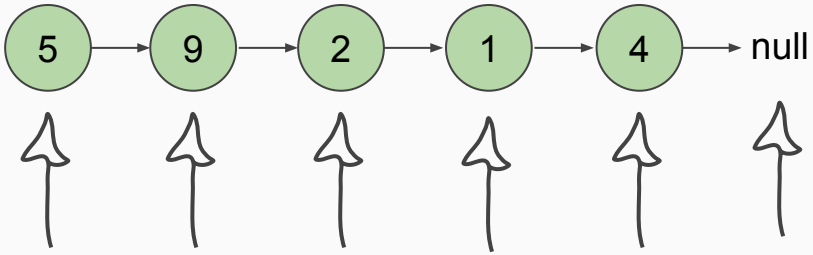
Lists and Functional Interfaces

Outline

- Lists
  - Recursion with rest / cons
- Generic Types
- Functional Interfaces
  - Higher Order Functions
- Filter, Map, Reduce

findMin - conceptual



Minimum so far: ~~5~~ ~~9~~ 2 ✅

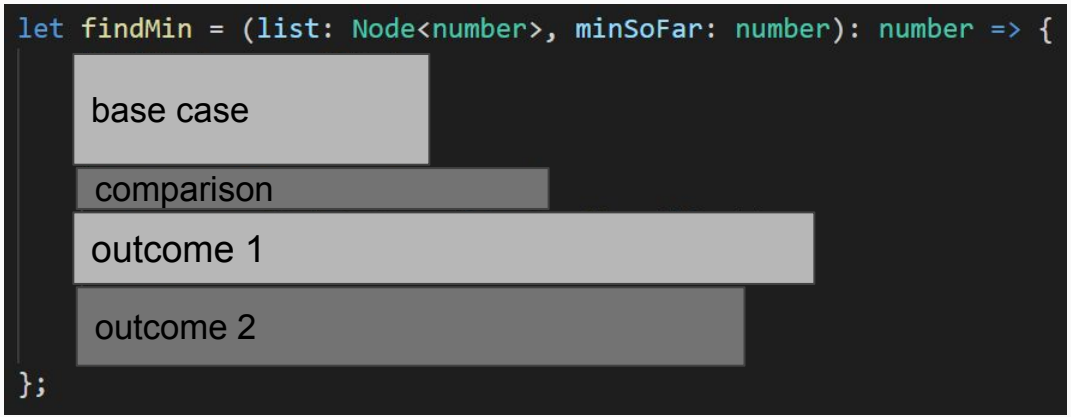
We knew we were done when we…

reached null!

When we weren't done, we…

1) compared current node's value to minimum so far
2) if smaller, updated minimum so far and moved to next node
3) if larger, just moved to next node

findMin - code!



We knew we were done when we…

reached null!

When we weren't done, we…

1) compared current node's value to minimum so far
2) if smaller, updated minimum so far and moved to next node
3) if larger, just moved to next node

findMin - code!

```
let findMin = (list: Node<number>, minSoFar: number): number => {
    if (list === null) {
        return minSoFar;
    }
    if (first(list) < minSoFar) {
        return findMin(rest(list), first(list));
    } else {
        return findMin(rest(list), minSoFar);
    }
};
```

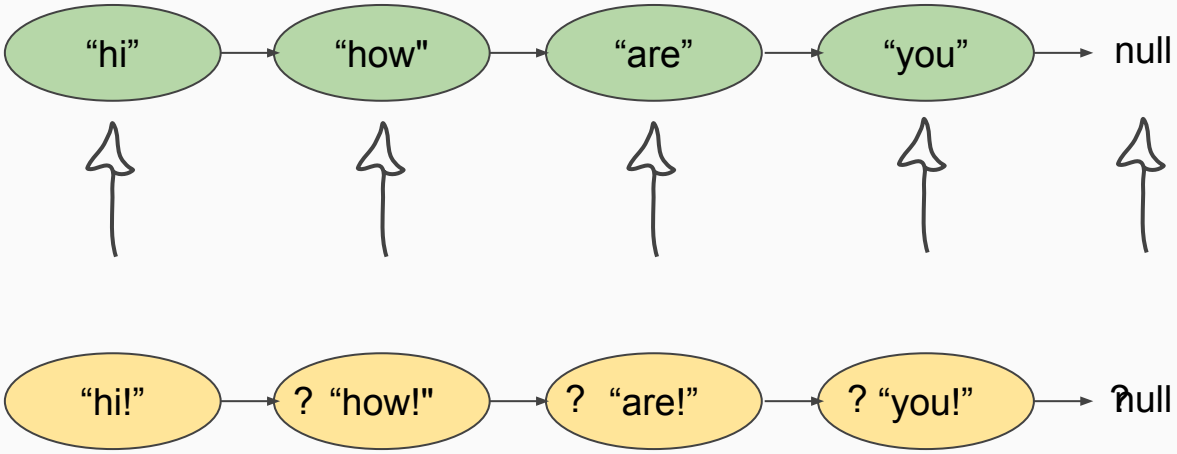
We knew we were done when we…

reached null!

When we weren't done, we…

1) compared current node's value to minimum so far
2) if smaller, updated minimum so far and moved to next node
3) if larger, just moved to next node

excite - conceptual



We knew we were done when we…

reached null!

When we weren’t done, we…

added our edited string onto the front of a new list

we don’t know what that list looks like yet… but our further calls to excite will tell us!

excite - code!

```
export let excite = (list: Node<string>): Node<string> => {
    base case
    recursive case
};
```

We knew we were done when we…

reached null!

When we weren't done, we…

added our edited string onto the front of a new list

we don't know what that list looks like yet… but our further calls to excite will tell us!

excite - code!

```
export let excite = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    }
    return cons(edited string, list to append to);
};
```

We knew we were done when we…

reached null!

When we weren't done, we…

added our edited string onto the front of a new list

we don't know what that list looks like yet… but our further calls to excite will tell us!

Generics

```
let matchingNum = (a: number, b: number): boolean => {
    return a === b;
};

let matchingString = (a: string, b: string): boolean => {
    return a === b;
};

let matchingBoolean = (a: boolean, b: boolean): boolean => {
    return a === b;
};
```

Generics

```
let matchingANY = <T> (a: T, b: T): boolean => {
    return a === b;
};
```

- Then you can use T as a type
- Consistent throughout function

Generics with Nodes

Data is a generic type

Can a node of one type “point to” a node of another type?

Consistency!

```
class Node<T> {
    data: T;
    next: Node<T> = null;
}
```

Check-in and Hot Date

Check-in Code: **`06E0C`**

Talk to your neighbor about whether or not it’s okay to listen to Holiday music yet!

Types

There’s value types: string, number, boolean

Reference types: Arrays, Objects

And “on the side” you also have functions

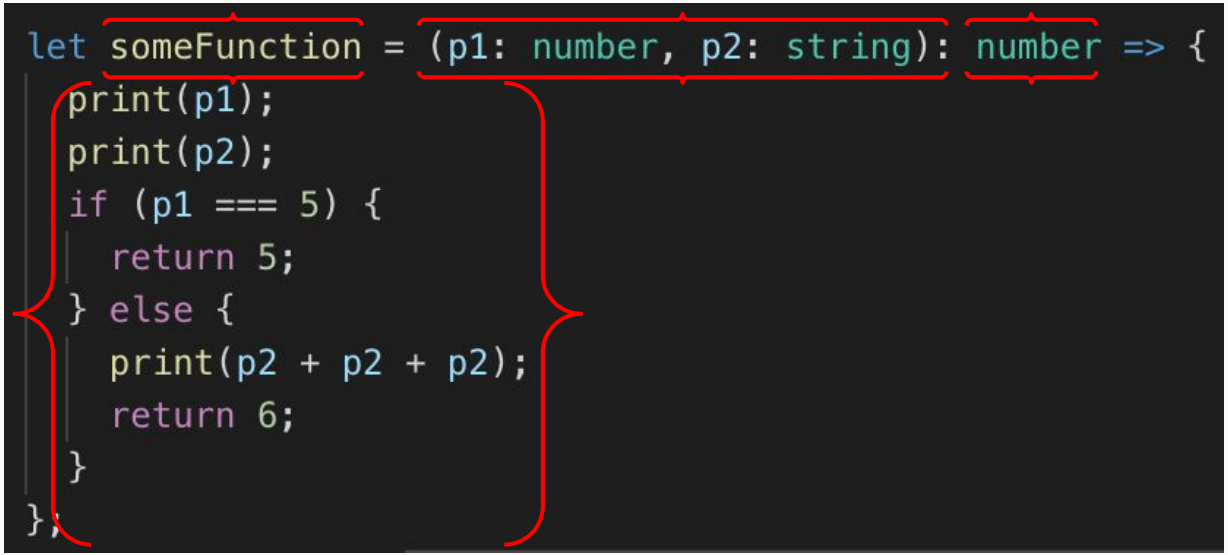
Functions

Do they have a type?

If they did have a type, what could we broadly categorize a function with?

Functions cont.

Parts of a function



Functions cont.

We can define a functions type based off its parameters and its return type.

We can formally specify this by using an interface.

Interfaces

We can define a functions type using an interface

Comma separated list of parameters, and a return type.

```
interface Name {
  (p1: type, p2: type): returnType;
}
```

Functional Interfaces

We can give a function's type a name by using an **interface**.

```
interface Mather {
  (n: number): number;
}
```

```
let inc: Mather = (n: number): number => {
  return n + 1;
};
```

```
let dec: Mather = (n: number): number => {
  return n - 1;
};
```

Higher order (DoMath)

```
interface Mather {
  (n: number): number;
}
```

```
let inc: Mather = (n: number): number => {
  return n + 1;
};

let dec: Mather = (n: number): number => {
  return n - 1;
};

let pow: Mather = (n: number): number => {
  return n * n;
};

let doMath = (num: number, func: Mather): number => {
  return func(num);
};
```

```
export let main = async () => {
  let num = 10;
  print(doMath(num, inc));
  print(doMath(num, dec));
  print(doMath(num, pow));
};
```

11
number

9
number

100
number

Predicate

Special functional interface that defines a function that takes in an element and returns a boolean value.

```
interface Predicate<T> {
    (item: T): boolean;
}
```

Filtering

The process of taking in a list and outputting another list that conforms to some criteria.

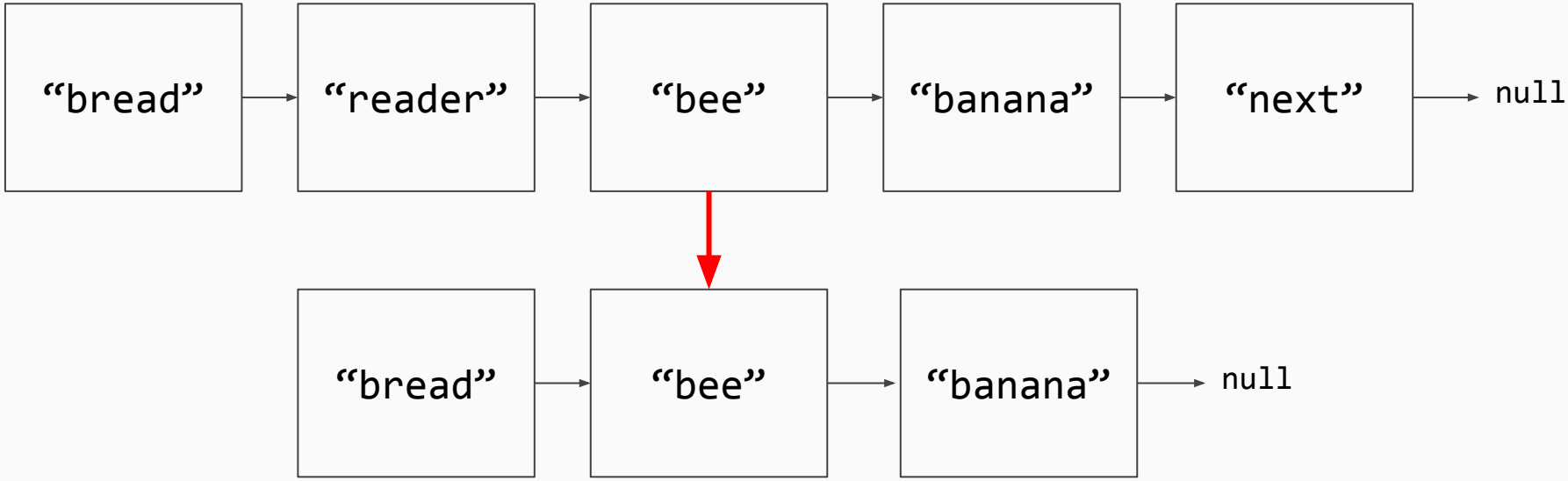
Ex.

Take in a number list and return a list of only numbers greater than five.
Take in a string list and return a list of only strings starting with “b”

Filtering

```
let filter = <T>(list: Node<T>, f: Predicate<T>): Node<T> => {
  if (list === null) {
    return null;
  } else if (f(first(list))) {
    return cons(first(list), filter(rest(list), f));
  } else {
    return filter(rest(list), f);
  }
};
```

Filtering



Transform

Special functional interface that defines a function that takes in an element and returns an element of another type.

`T` and `U` can be the same type!

```
interface Transform<T, U> {
    (item: T): U;
}
```

Mapping

The process of taking in a list and outputting another list where every element has been “transformed” into another element.

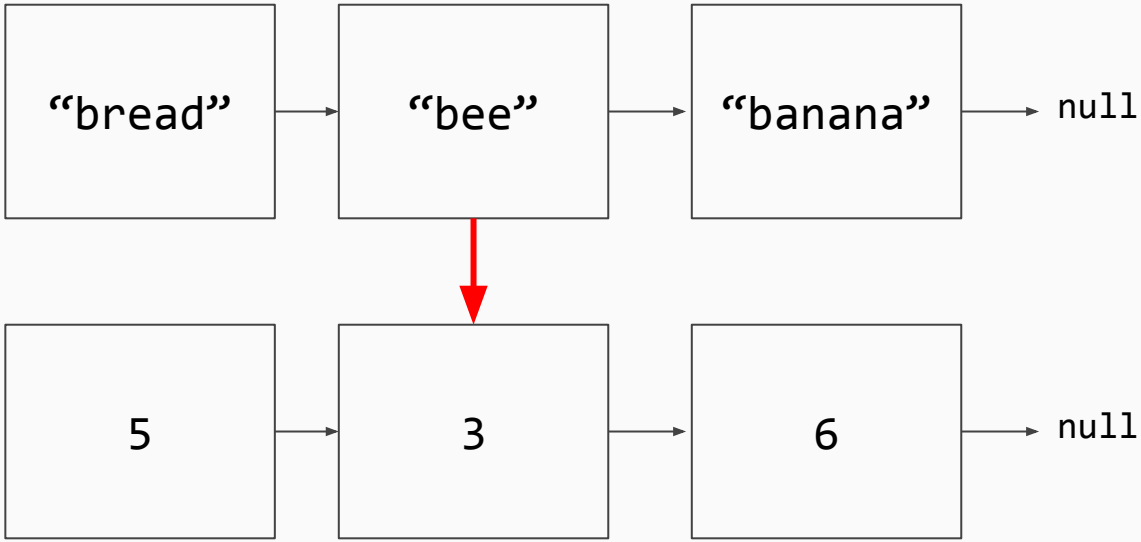
Ex.

Take in a number list and return a list with every number squared.
Take in a string list and return a list of numbers that correspond to a string’s length
Take in a boolean list and return a list with every value flipped

Mapping

```
let map = <T, U> (xs: Node<T>, f: Transform<T, U>): Node<U> => {
    if (xs === null) {
        return null;
    } else {
        return cons(f(first(xs)), map(rest(xs), f));
    }
};
```

Mapping



Reducer

Special functional interface that defines a function that takes in two elements (a memo and an item) and reduces a list to one value.

```
interface Reducer<T, U> {
    (memo: U, item: T): U;
}
```

Reduce

The process of taking in a list and outputting a single value after processing the entire list.

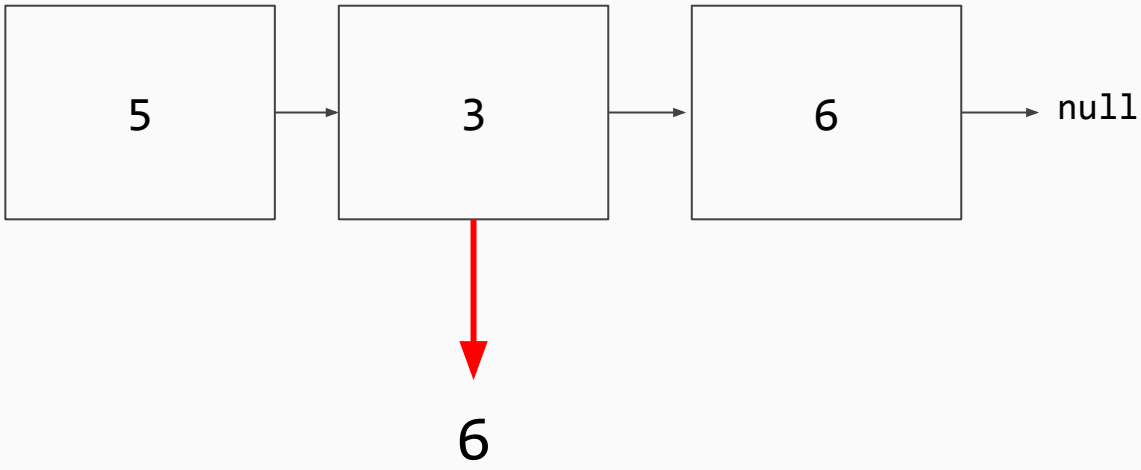
Ex.

Take in a number list and return the highest number in the list.
Take in a string list and return all of the strings concatenated together in order.
Take in a number list and return true if every number is 5

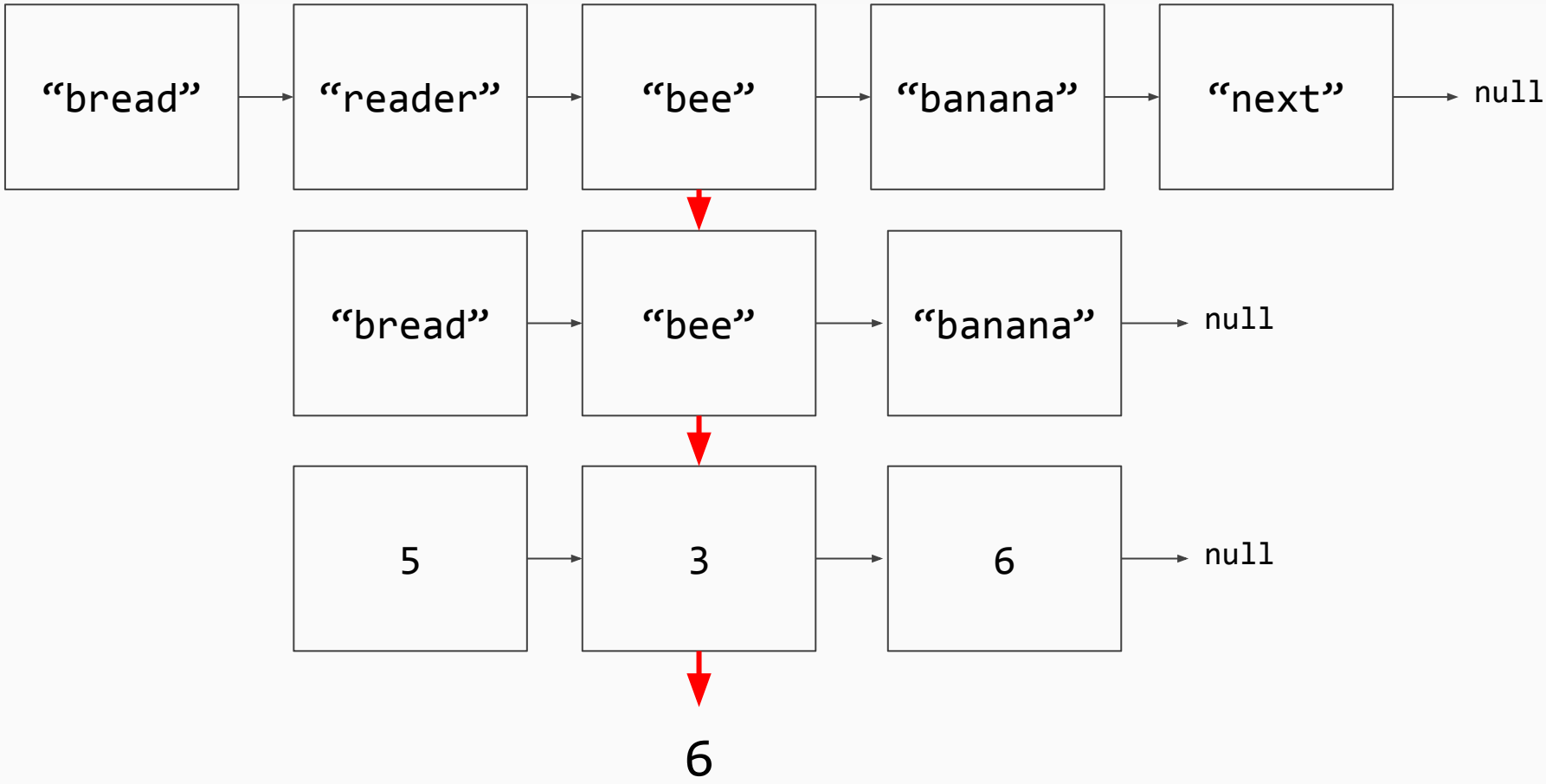
Reduce

```
let reduce = <T, U> (xs: List<T>, f: Reducer<T, U>, memo: U): U => {
   if (xs === null) {
      return memo;
   } else {
      return reduce(rest(xs), f, f(memo, first(xs)));
   }
};
```

Reduce

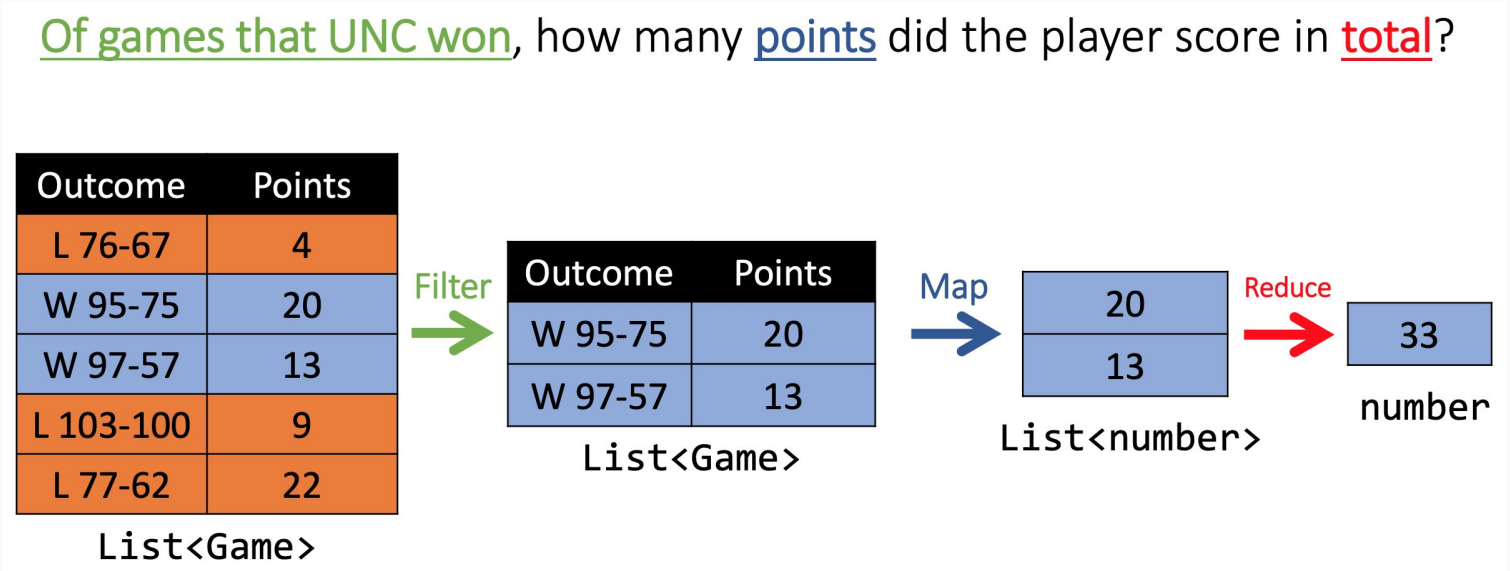


Filter, Map, Reduce



Filter, Map, Reduce

Fundamental pattern / pipeline for data processing



Filter, Map, Reduce

Of games that UNC won, how many points did the player score in total?

Outcome	Points
L 76-67	4
W 95-75	20
W 97-57	13
L 103-100	9
L 77-62	22

List<Game>

Filter →

Outcome	Points
W 95-75	20
W 97-57	13

List<Game>

Map →

20
13

List<number>

Reduce →

33

number

```
let list =
```

```
reduce(map(filter(list, isWin), gameToPoints), sum, 0);
```

Type Inference

When you declare a function with a definition that matches an interface, TypeScript can infer that that function is of that type.

What type of function is this?

```
let x = (n) => {
  return n > 5
}
```

Function Literals

We can define functions ***inline*** for use with higher order functions.

Take the inc function for example

```
let inc: Mather = (n: number): number => {
  return n + 1;
};
```

Function Literals

We can define functions ***inline*** for use with higher order functions.

Take the inc function for example

```
let inc: Mather = (n: number): number => {
  return n + 1;
};
```

Function Literals

We can define functions ***inline*** for use with higher order functions.

Take the inc function for example

```
let inc: Mather = (n: number): number => {
  return n + 1;
};
```

Function Literals

We can define functions ***inline*** for use with higher order functions.

Take the inc function for example: Literal!

```
                          (n: number): number => {
  return n + 1;
};
```

Function Literals

We can define functions ***inline*** for use with higher order functions.

Take the inc function for example: Literal!

```
(n: number): number => {
  return n + 1;
};
```

Function Literals

We can define functions ***inline*** for use with higher order functions.

Take the inc function for example: Literal!

```
(n: number) => {
  return n + 1;
};
```

Function Literals

We can define functions ***inline*** for use with higher order functions.

Take the inc function for example: Literal!

```
(n: number) => { return n + 1; };
```

Function Literals

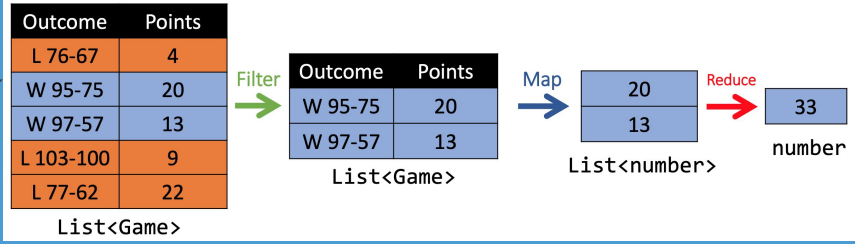
We can define functions ***inline*** for use with higher order functions.

Take the inc function for example: Literal!

```
(n: number) => n + 1;
```

Filter, Map, Reduce

Of games that UNC won, how many points did the player score in total?




```
let list = [redacted]
let wins = filter(list, (g) => g.uncPoints > g.opponentPoints);
let points = map(wins, (g) => g.uncPoints);
let allPoints = reduce(points, (m, n) => m + n, 0);
```

```
reduce(map(filter(list, (g) => g.uncPoints > g.opponentPoints), (g) => g.uncPoints), (m, n) => m + n, 0);
```