`map` & `reduce`

Lecture 21 – COMP110 – Spring 2019

Announcements

- PS6 – Compstagram – Due Friday 4/26
- WS5 – Higher-Order Functions – Posts Tonight
    - Since there's no quiz on Higher-order Functions this is important practice.
- Apply to be a UTA
    - Link on home page
    - Applications due April 28^{th} at 11:59pm

Final Exams

- Section 1 – Monday, May 6th at 12pm

- Section 2 – Tuesday, April 30th at 4pm

- Seat assignments will be reshuffled for final.

- Review sessions will be held for each section (time and location TBD).
  - Office hours close for the semester on LDOC at 5pm.

- 3-in-24 hours Pink Slip?
  - Makeup registration form, pink slip upload, and makeup time on home page.

1. Which of these functions is an implementation of the Transform<T, U> functional interface?

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

```
A: (m: number, n: number): number => { /* ... */ };
B: (m: number): number => { /* ... */ };
C: (m: string): number => { /* ... */ };
D: (m: string): string => { /* ... */ };
E: None of the above
F: All except A
G: All of the above
```

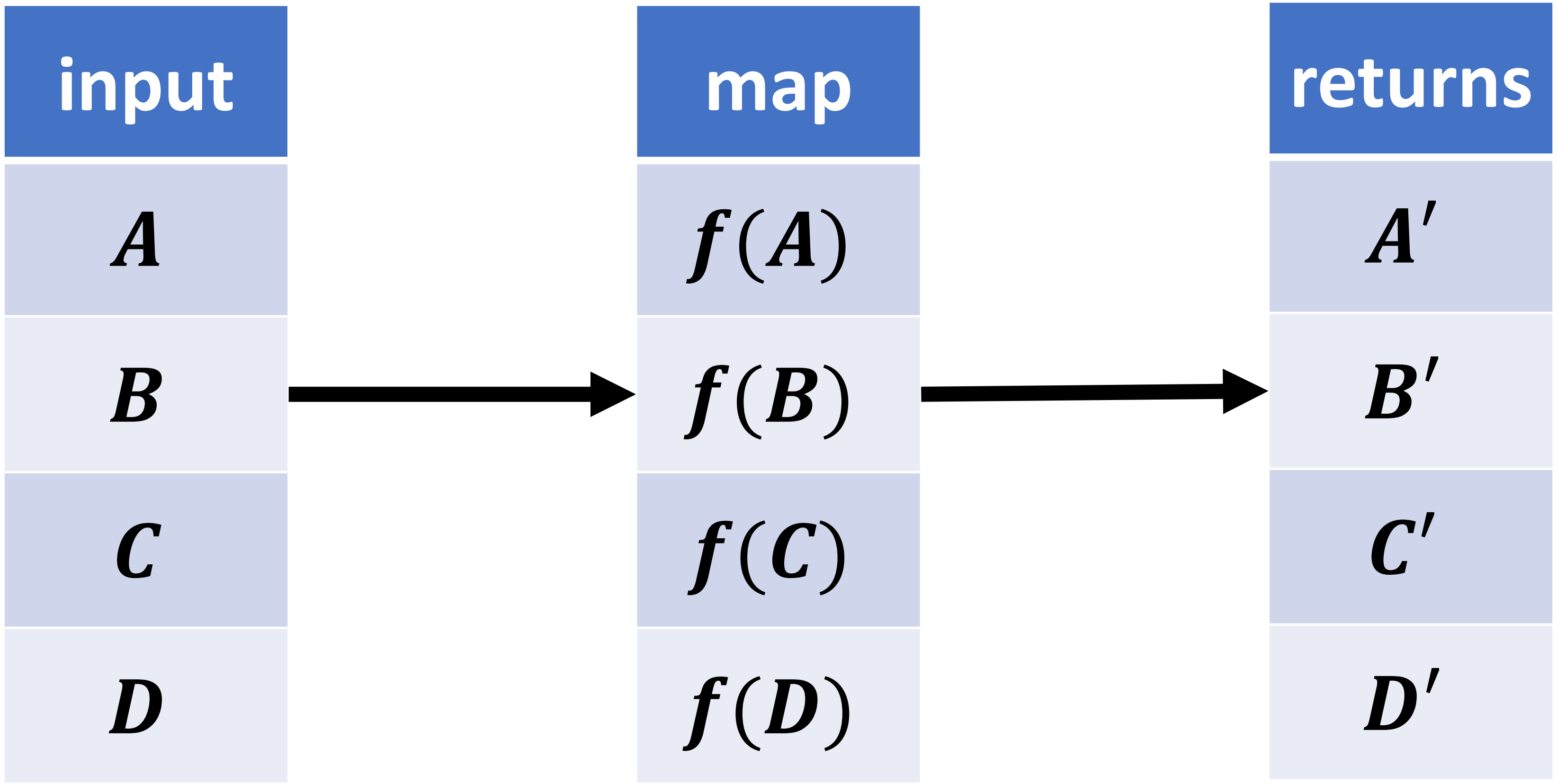
2. Given the definitions on the left, after the code below completes, what is the result variable's type and what is its value?

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

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

let strToLen: Transform<string, number> = (s: string): number => {
    return s.length;
};
```

```
let words: Node<string> = listify("lets", "go", "unc");
let result: ____________ = map(words, strToLen);
```

The `map` Function

Given an *input* list and a *transform function* f, returns a new list with f applied to every element in the input list.



The **`Transform<T, U>`** Functional Interface

- What if we wanted to generically describe a function that given an argument of any type ***T*** returned a value of any type ***U***?

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

- Examples:
  1. a function that takes an argument of type string and returns a number
  2. takes string and returns a string... *it's ok for T and U to be the same type!*

1

```
(s: string): number => {
      return s.length;
}
```

2

```
(s: string): string => {
   return s.toUpperCase();
}
```

An implementation of **map**

```
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));
   }
};
```

- Notice this is the recursive List building pattern we've used all along, just with the function *f* being passed in and called on each element as we *cons* it onto the resulting List.

Hands-on: Writing a Transform function and using map

- Open 00-map-app.ts
- Goal: After loading Berry's game data from data/joel-berry-ii.csv, produce a list of *only* Berry's points per game values

1. **TODO #1)** Declare a function named ***gameToPoints*** that is given a ***Game*** object named ***g*** as a parameter and returns a ***number***. It should simply return the ***points*** property of the Game parameter: ***g.points***
2. **TODO #2)** Call the map function using the function defined in #1:
   `map(games, gamesToPoints)`

- You should see a list of points values printed after loading your data. Can you trace through how the map function is calling `gameToPoints`?
- Check-in on PollEv.com/compunc when this is working

```
// TODO #1: Define a function named gameToPoints
// It should take in a Game object as a parameter and return a number
// The number it returns should be the game's points property
let gameToPoints = (g: Game): number => {
    return g.points;
};
```

```
// TODO #2 - Assign to points the result of calling map with
// the games List and the gameToPoints function you wrote below.
points = map(games, gameToPoints);
```

Follow-along: Anonymous Functions

- Previously we introduced anonymous functions that relied on type inference to infer the parameter and return types, let's apply that same technique here...

- Open 01-map-shorthand-app.ts

```
let points = map(games, (g) => { return g.points; });
```

Shorthand Function Literals

- **IF, and only if, you are writing a function whose body contains only a single return statement**, like this function literal:

```
(s) => { return s.length > 3; }
```

- Then, you can rewrite the function using shorthand syntax. This syntactical change:
    1. Drops the curly braces
    2. Drops the return keyword
    3. Drops the semi-colon following the return statement's expression

```
(s) => s.length > 3
```

3. Which of these functions is an implementation of the **`Reducer<T, U>`** functional interface?

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

```
A: (m: number, n: number): number => { /* ... */ };
B: (m: number, n: string): number => { /* ... */ };
C: (m: string, n: number): number => { /* ... */ };
D: (m: string, n: string): string => { /* ... */ };
E: All except B
F: All except C
```

4. What is the output?

```
let add = (m: number, n: number): number => {
  return m + n;
};

let xs: Node<number> = listify(3, 4, 5);
let s: number = 0;
s = add(s, first(xs));
s = add(s, first(rest(xs)));
s = add(s, first(rest(rest(xs))));
print(s);
```

The `reduce` Function

Given an *input* list, a *reducer function f*, and an initial *memo(ry)* value, **reduce** gives *f* the memo and the next value. Whatever f returns is used as the next memo for the next element until the final value returned is the solution.

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

The `reduce` Function's Intuition

List: `1 → 2 → 3 → null`

How can we reduce a list using the following *add* reducer?

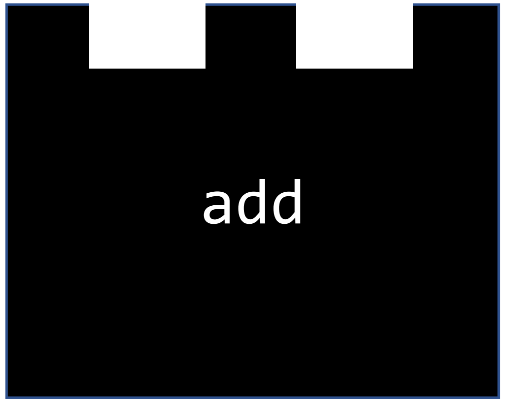
```
let add = (memo: number, item: number): number => {
    return memo + item;
};
```

It's like scanning down a list and keeping track of some ***"reduced"*** or "accumulated" value (like a sum) as you continue each step of the way...

The `reduce` Function's Intuition

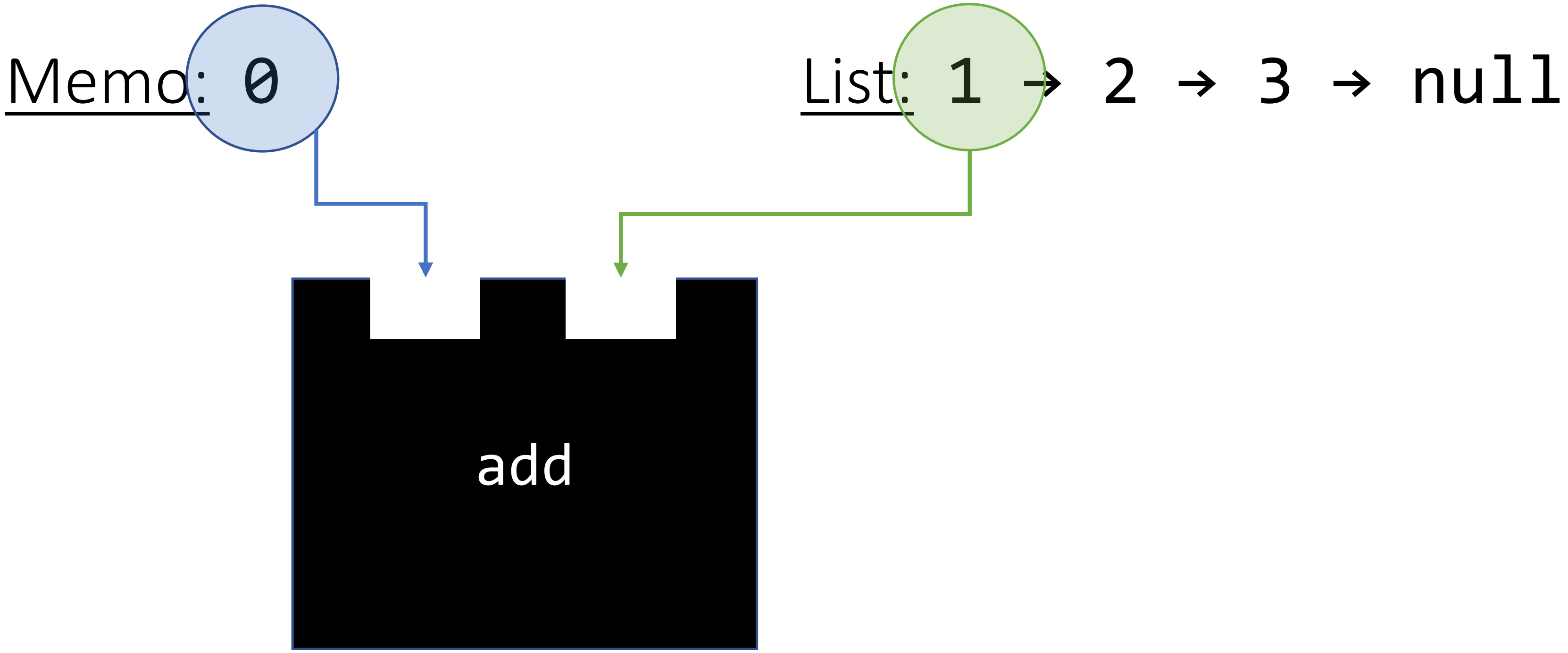
Memo: `0`

List: `1 → 2 → 3 → null`



Reduce's initial "memo" is the starting value. Here, since we're trying to add up all the numbers, we'll start with a memo of 0.

The `reduce` Function's Intuition

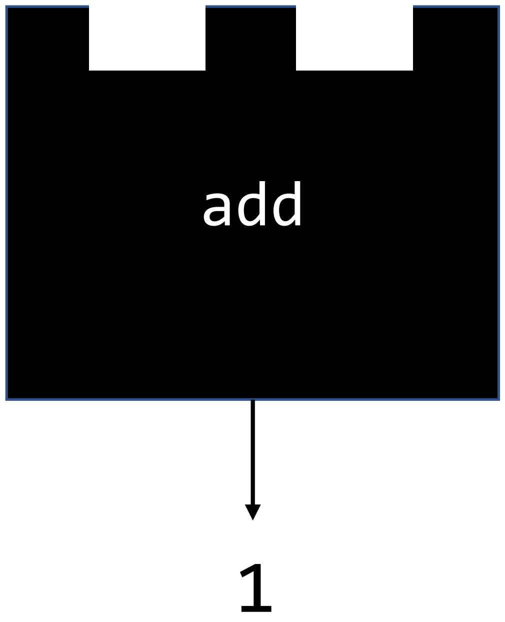


The reduce function calls add (the reducer) with memo and the next value from the list.

The `reduce` Function's Intuition

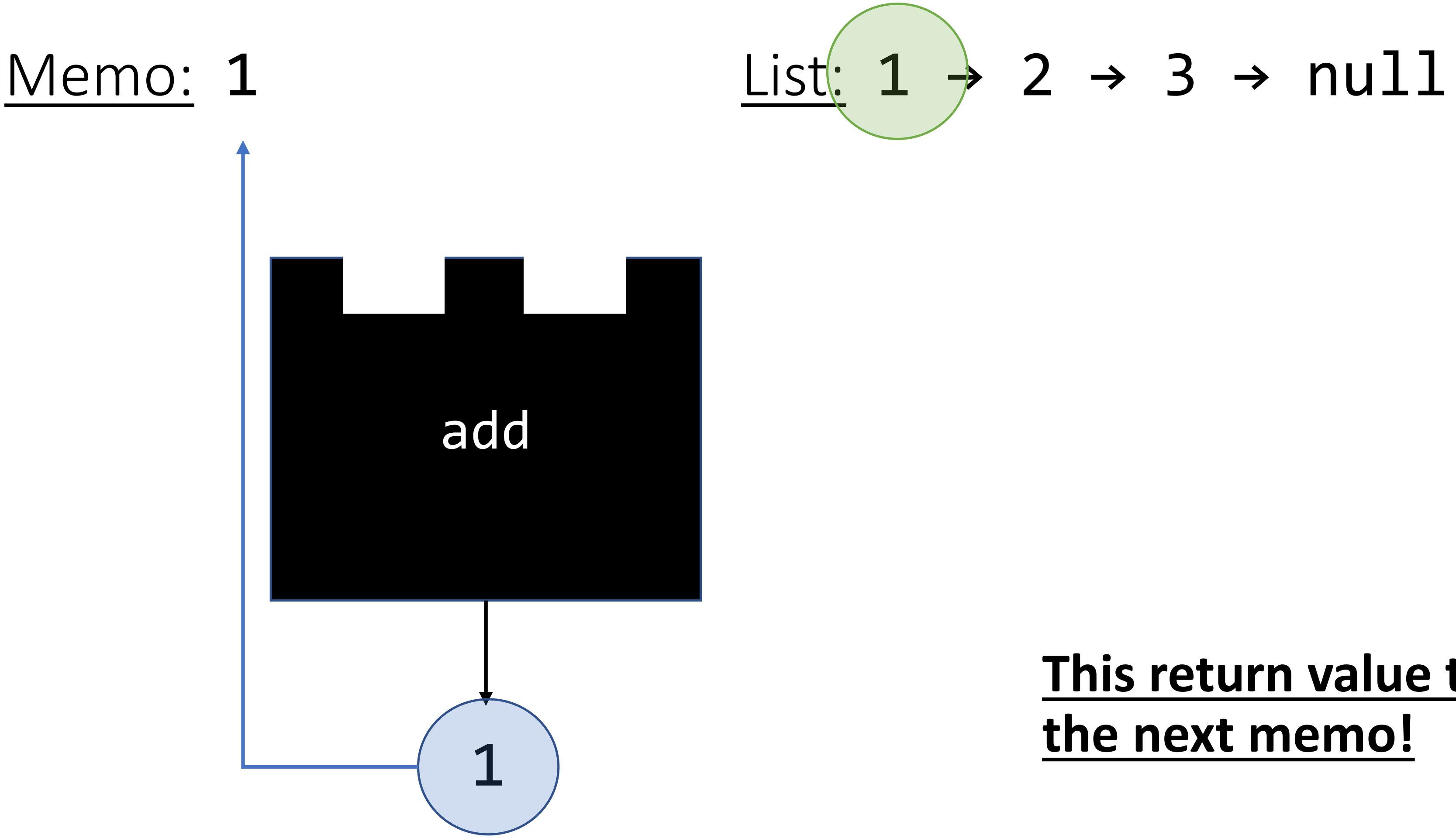
Memo: `0`

List: `1 → 2 → 3 → null`



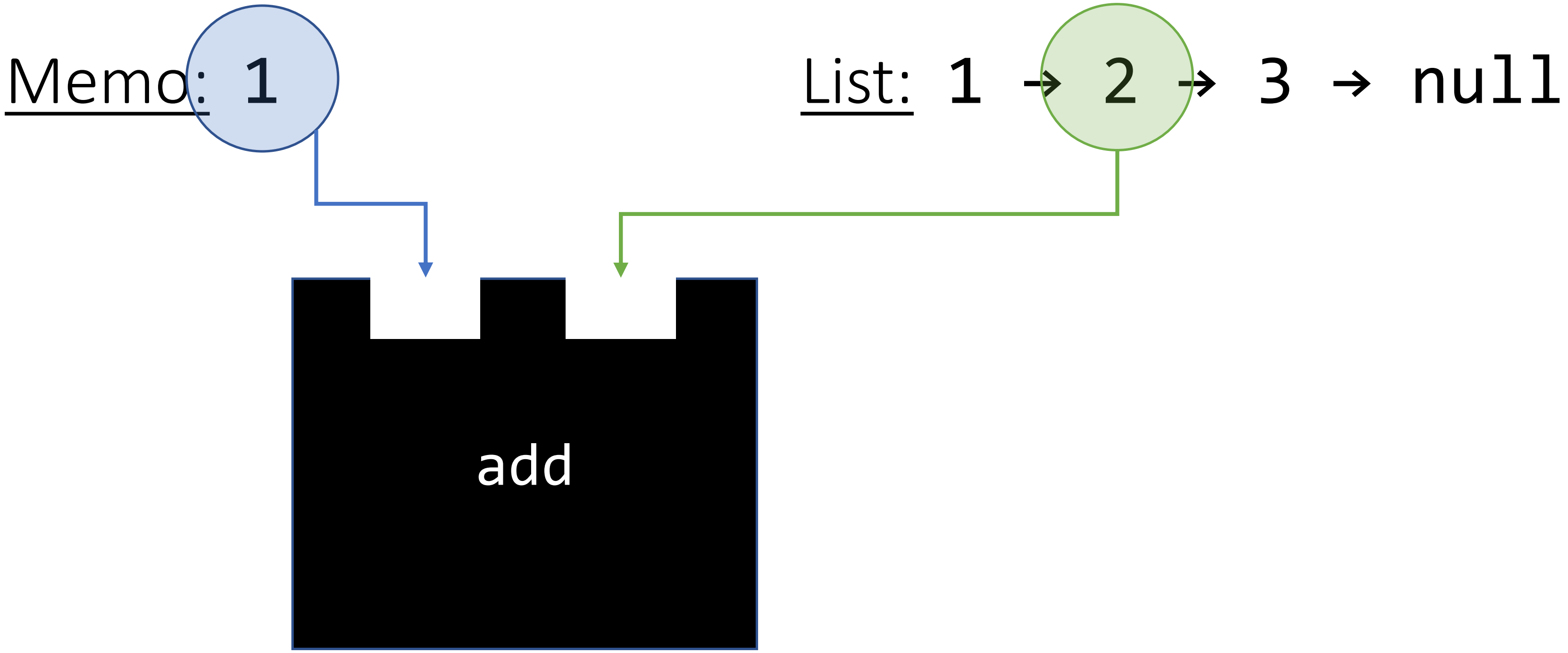
The reducer produces a return value.

The `reduce` Function's Intuition



This return value then becomes the next memo!

The `reduce` Function's Intuition

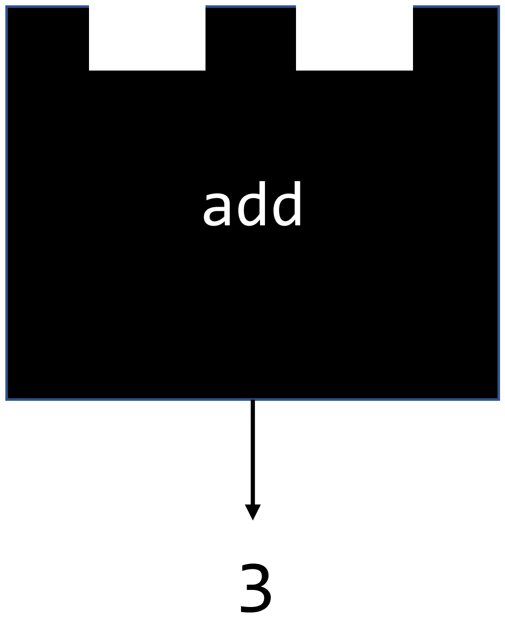


The reduce function calls add (the reducer) with memo and the next value from the list.

The `reduce` Function's Intuition

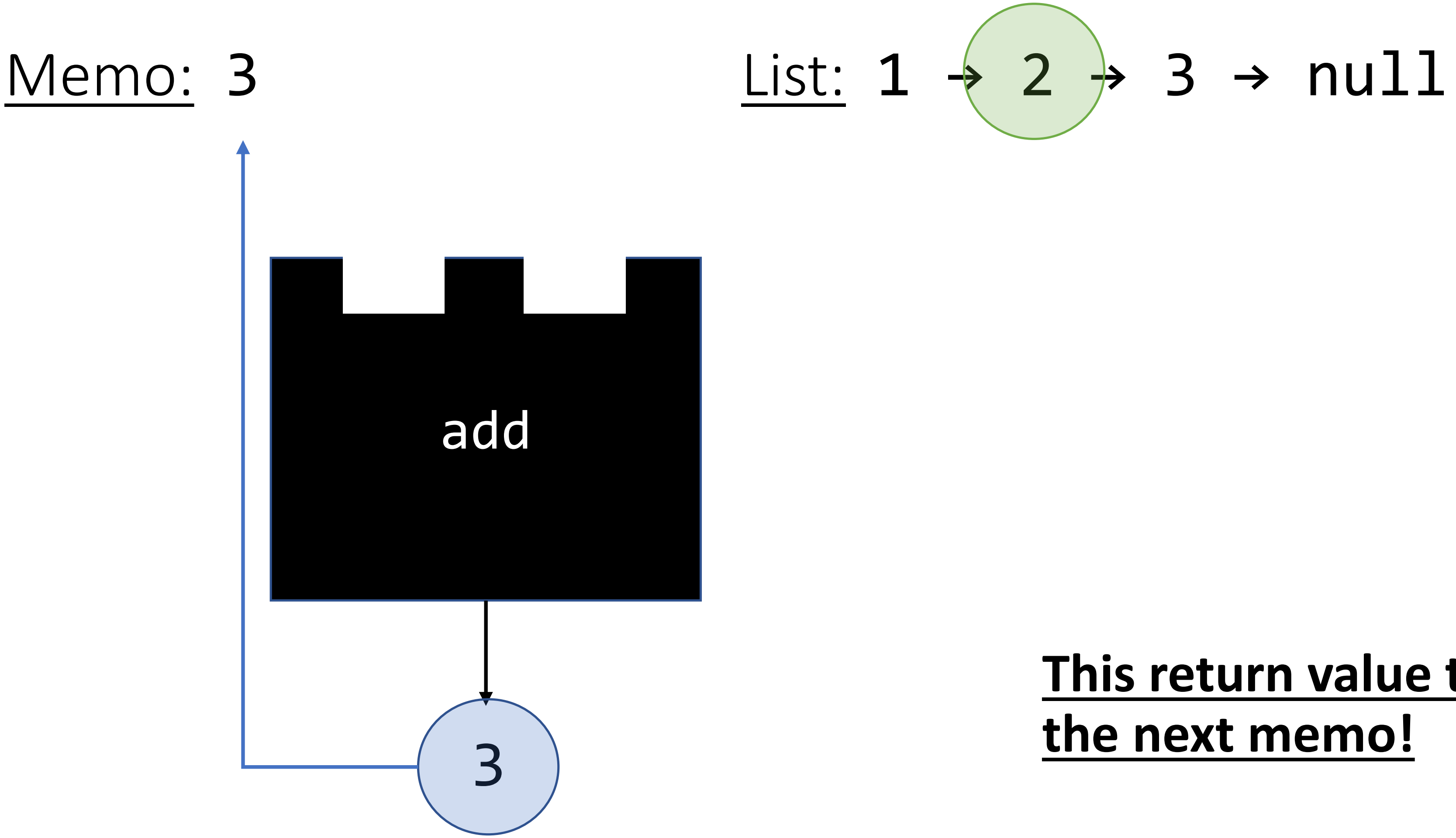
Memo: `1`

List: `1 → 2 → 3 → null`



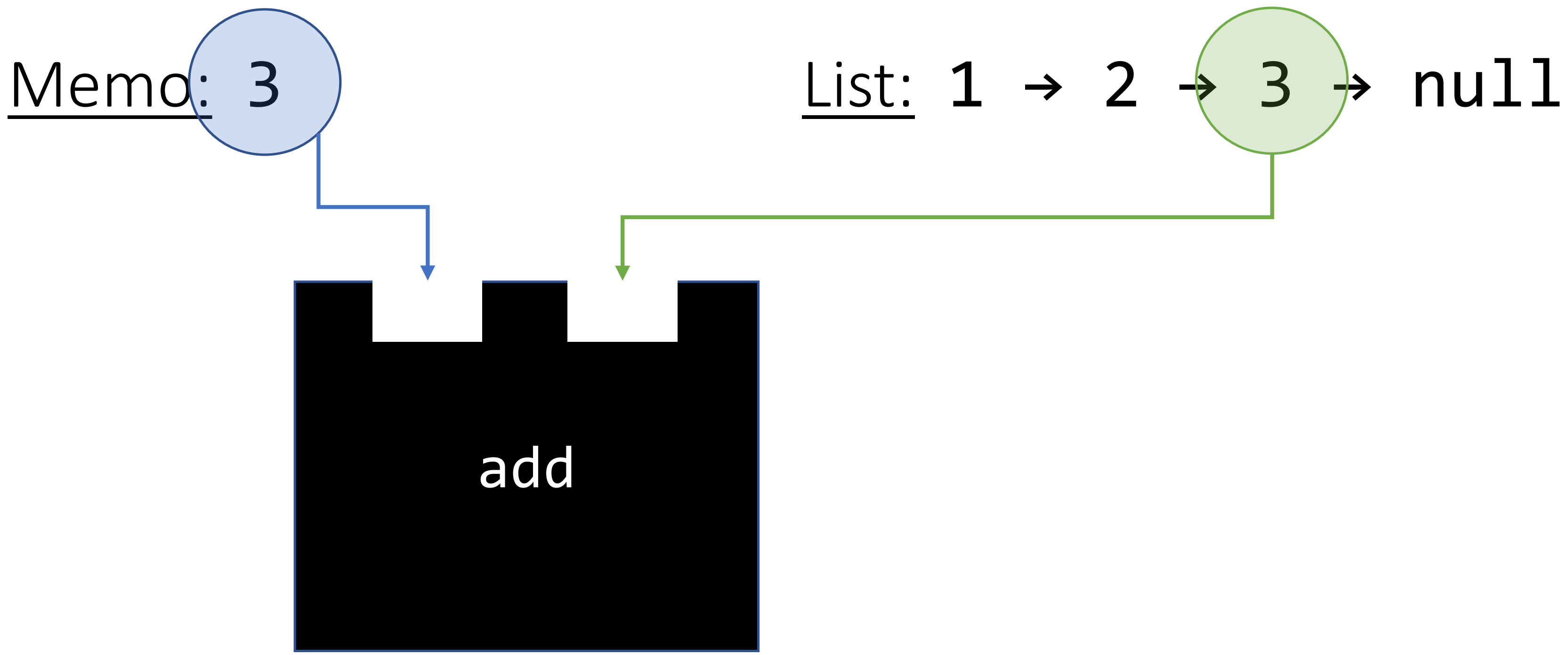
The reducer produces a return value.

The `reduce` Function's Intuition



This return value then becomes the next memo!

The `reduce` Function's Intuition

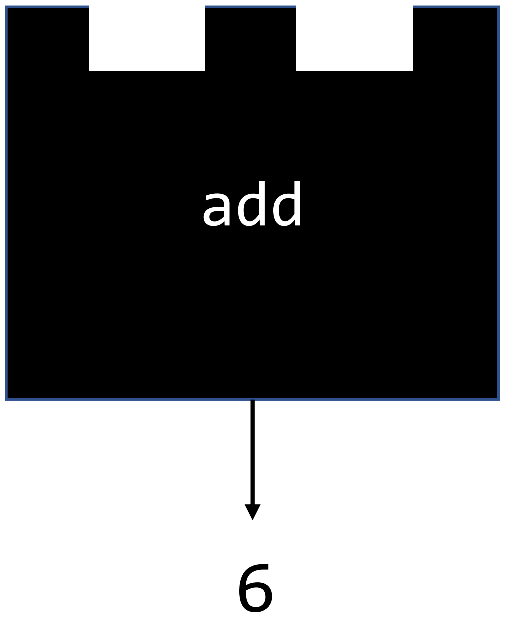


The reduce function calls add (the reducer) with memo and the next value from the list.

The `reduce` Function's Intuition

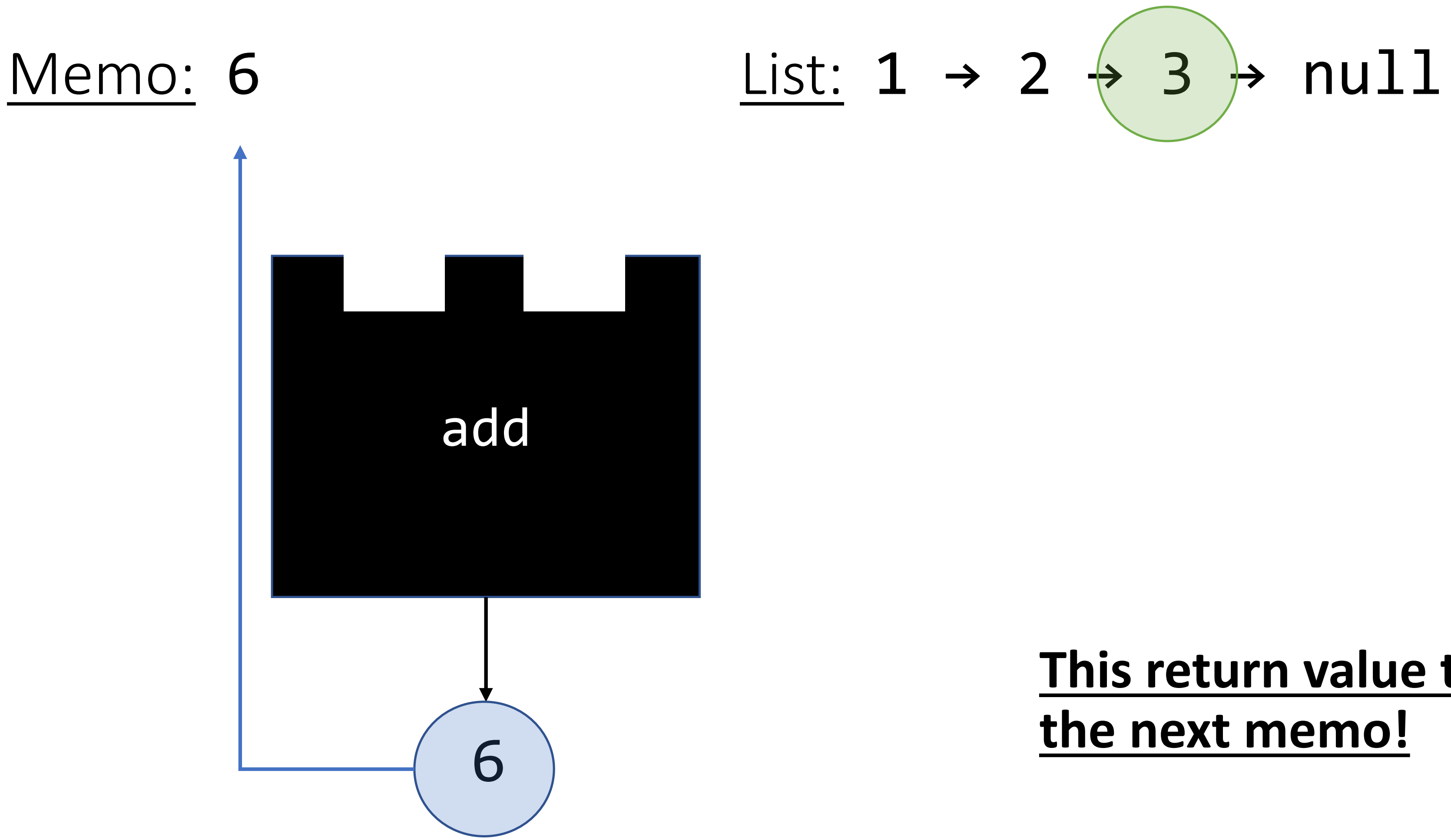
Memo: 3

List: `1 → 2 → 3 → null`



The reducer produces a return value.

The `reduce` Function's Intuition

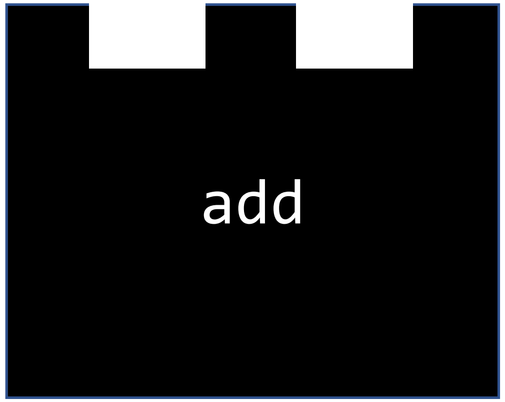


<u>This return value then becomes the next memo!</u>

The `reduce` Function's Intuition

Memo: 6

List: 1 → 2 → 3 → `null`



When the end of the list is reached, reduce's returned value is **memo**.

The **reduce** Function's Intuition

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

```
let add = (memo: number, item: number): number => {
    return memo + item;
};
```

```
reduce(1→2→3→null, add, 0)
    reduce(2→3→null, add, 1)
        reduce(3→null, add, 3)
            reduce(null, add, 6)
                6
```

Hands-on: Writing a Reducer function and using **reduce**

- Open 02-reduce-app.ts

- Goal: After loading Berry's game data from data/joel-berry-ii.csv, find the most points Joel scored in a game

1. **TODO #1)** Declare a function named ***max*** that is given two number parameters and returns the larger of the two

2. **TODO #2)** In the main function, **assign** to the variable ***high*** the result reducing:
   `reduce(points, max, 0)`

- You should see the season high points printed out after loading the data.

- Check-in on PollEv.com/compunc when this is working

```
// TODO #1 - Write a reducer named max that is given two numbers
// and will return the larger of the two numbers.
let max = (m: number, n: number): number => {
    if (m > n) {
        return m;
    } else {
        return n;
    }
};
```

```
// TODO #2 - Assign to high the result of calling reduce with arguments
// 1. the points array
// 2. your max reducer function
// 3. an initial memo value of 0
let high: number = reduce(points, max, 0);
```

Array's `filter`, `map`, and `reduce` Methods

- Like `string` values, arrays have built-in methods

- Among other methods, arrays have three other built-in higher-order methods:

1. filter
2. map
3. reduce

Array's **filter** Method

- Every array of type **T[]** has a **filter** method.

- The **filter** method has a single parameter: a **Predicate<T>** of the same type **T**

- For example:

```
let a = [-1, 0, 1, 2];
let b = a.filter((x) => x > 0);
print(b); // Prints: 1, 2
```

- Calling the **filter** method on array **a** will return a new array of type **T**.
  The filter method tests all elements in the original array *using the Predicate<T>*.
  Elements that return true will be copied to the returned array.

Array's **map** Method

- Every array of type **T[]** has a **map** method.

- The **map** method has a single parameter: a **Transform<T, U>** of the same type **T**
  - The **map** method will return an array of type **U[]**

- For example:

```
let a = ["one", "two", "three"];
let b = a.map((s) => s.length);
print(b); // Prints: 3, 3, 5
```

- Calling the **map** method on array **a** will return a new array of type **U[]**.
  The map method transforms all elements in the original array *using the Transform<T,U>*.
  All transformed elements are copied to the returned array in the same order.

Array's **reduce** Method

- Every array of type **T[]** has a **reduce** method.

- The **reduce** method has two parameters:
  1. a **Reducer<T, U>** of the same type **T**
  2. An initial **memo** ("memory" accumulator) value of type **U**

- For example:

```
let a = [1, 2, 3];
let b = a.reduce((memo, x) => memo + x, 0);
print(b); // Prints:6
```

- Calling the **reduce** method on array **a** will return a single value of type **U**. Starting with the initial **memo** parameter, it will call the reducer with memo and each element in **a** successively replacing memo's value with the reducer's returned value. The final **memo** value is returned.

Hands-on: filter/map/reduce Pipeline

- Open **`03-stats-app.ts`**

1. Assign to the **filtered** variable the result of calling the **filter** with the **games** List and one of **Predicate** functions below:

```
let filtered: Game[] = games.filter(PREDICATE);
```

2. Assign to the **values** variable, the result of calling **map** with the **filtered** List and one of the **Transform** functions below:

```
let values: number[] = filtered.map(TRANSFORM);
```

3. Assign to the result variable, the result of calling **reduce** with the **values** List and one of the **Reducer** functions below (what should the memo be?):

```
let result: number[] = values.reduce(REDUCER, INITIAL_MEMO);
```

4. Now change your code to find the max # of assists Joel Berry had in a game where he scored less than 15 points. Check-in on PollEv.com/compunc when you've got it.

```
// TODO #1
let filtered: Node<Game> = games.filter(fewPoints);
// TODO #2
let values: Node<number> = filtered.map(toAssists);
// TODO #3
let result: number = values.reduce(max, 0);
```

filter-map-reduce Pipeline

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

`Game[]`

Filter →

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

`Game[]`

Map →

20
13

`number[]`

Reduce →

33

`number`

filter-map-reduce Data Processing Pipeline

Of games

- **that UNC won**
- **that UNC lost**
- **with 3+ assists**
- **with a block**
- **etc**

, what was the

points	**total**
assists	**average**
fouls	**min**
blocks	**max**
etc	**etc**

```
Filter:   Game[]   →   Game[]
Map:      Game[]   →   number[]
Reduce:   number[] →   number
```

Big idea: We can **select any combo of a** filter, map, and reduce sequence.
Result: (# Predicates) x (# Transforms) x (# Reducers) different analyses.