`map`, `reduce`, and Type Inference

Lecture 16 – COMP110 – Fall 2018

Announcements

- Quiz 5 - Pushed back to Thursday, November 15th
  - We'll use Tuesday for more practice and review
- Next Worksheet - Out today, due Tuesday 11/13 at 11:59pm
- Next Problem Set - Basketball Data - Out Soon!
  - Due Tuesday 11/20 at 11:59pm

0. Fill in the correct types given the code below.

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

let aTest: Predicate<number> = (x: ______): ______ => {
  /* ... more code ... */
};
```

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
```

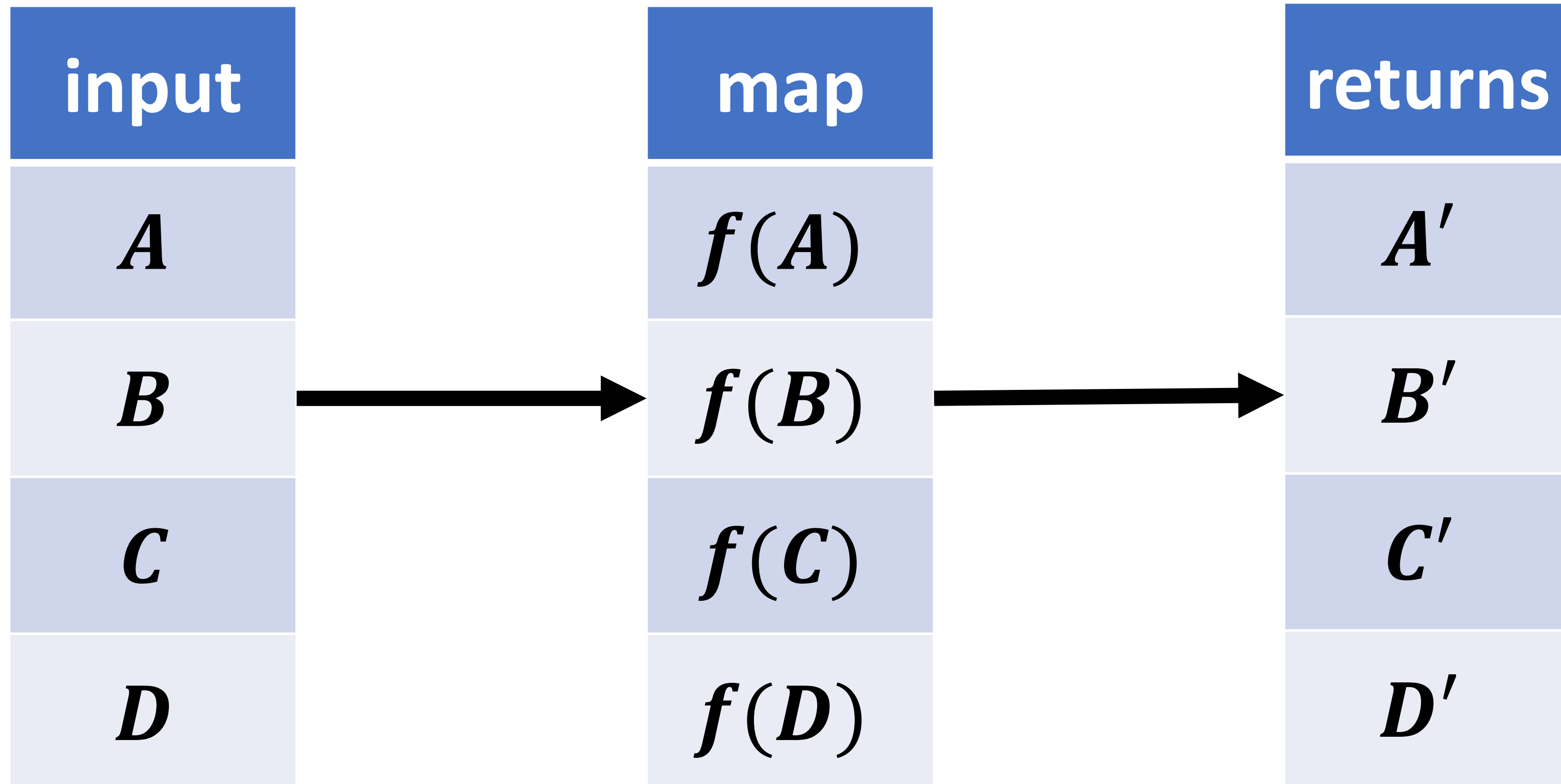
2. Given the definitions on the left, after the code below completes, what is the result variable's type and what is its value?

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

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

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!*

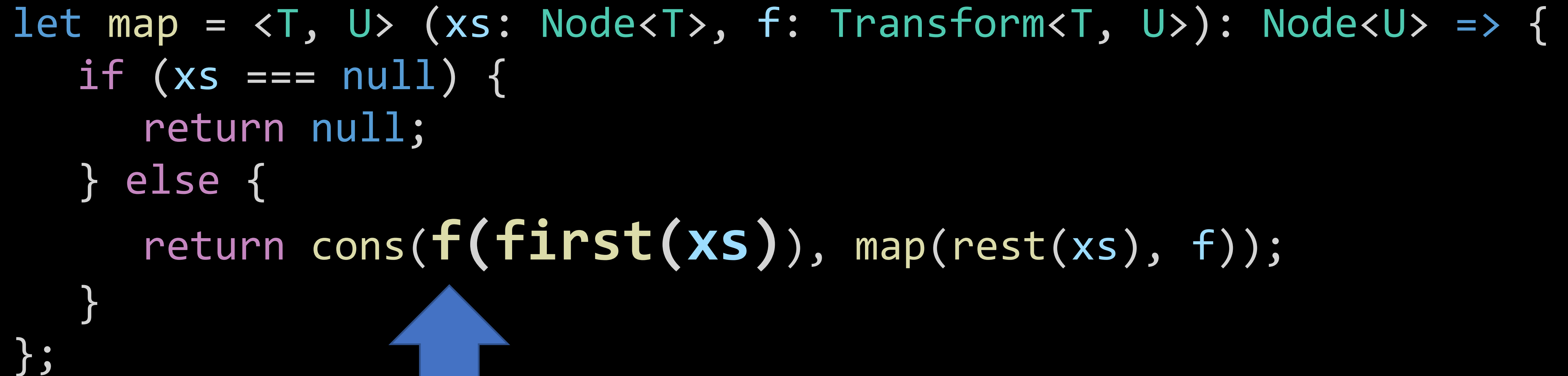
1

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

2

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

An implementation of **map**



- 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 lec16 / 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

- Last Thursday 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* 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: List<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?

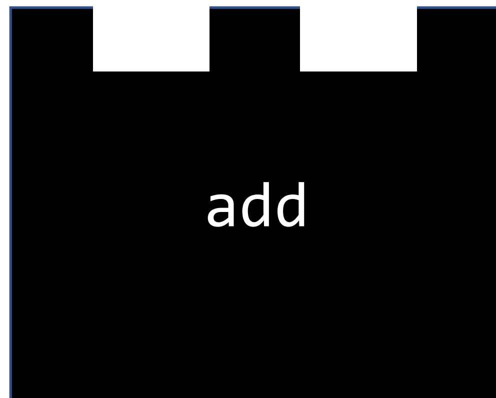
```
let add = (memo: number, item: number): number => {
    return memo + item;
};
```

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

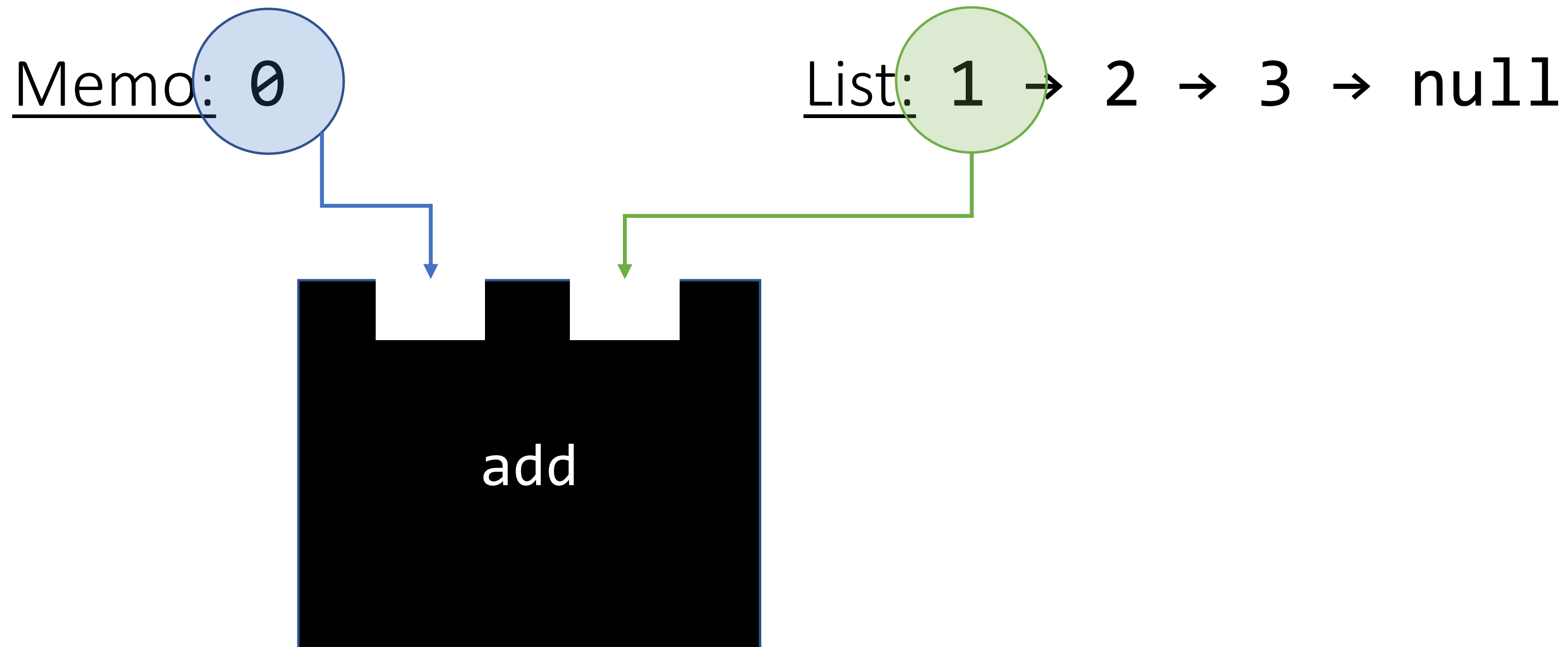
Memo: `0`

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

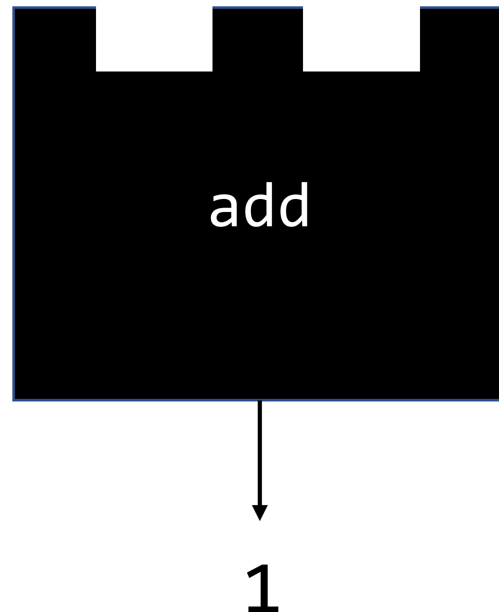


The reduce function calls add (the reducer) with memo and the next value from the list.

The `reduce` Function's Intuition

Memo: `0`

List: `1 → 2 → 3 → null`

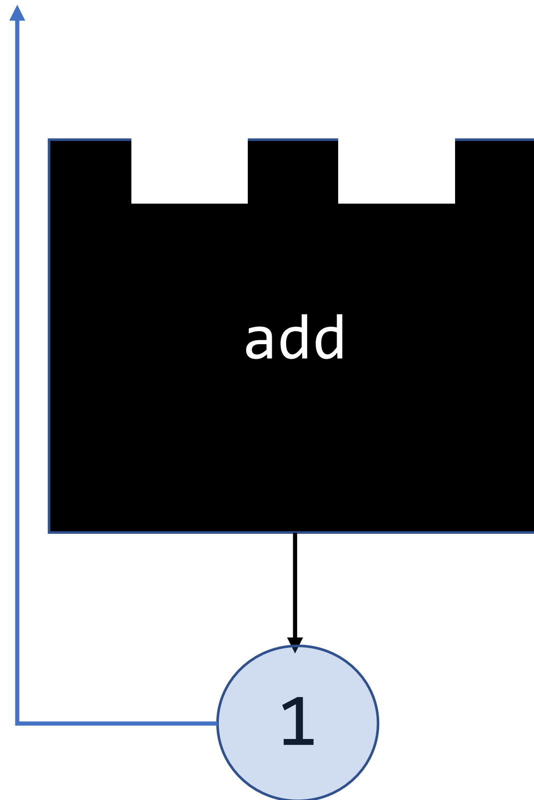


The reducer produces a return value.

The `reduce` Function's Intuition

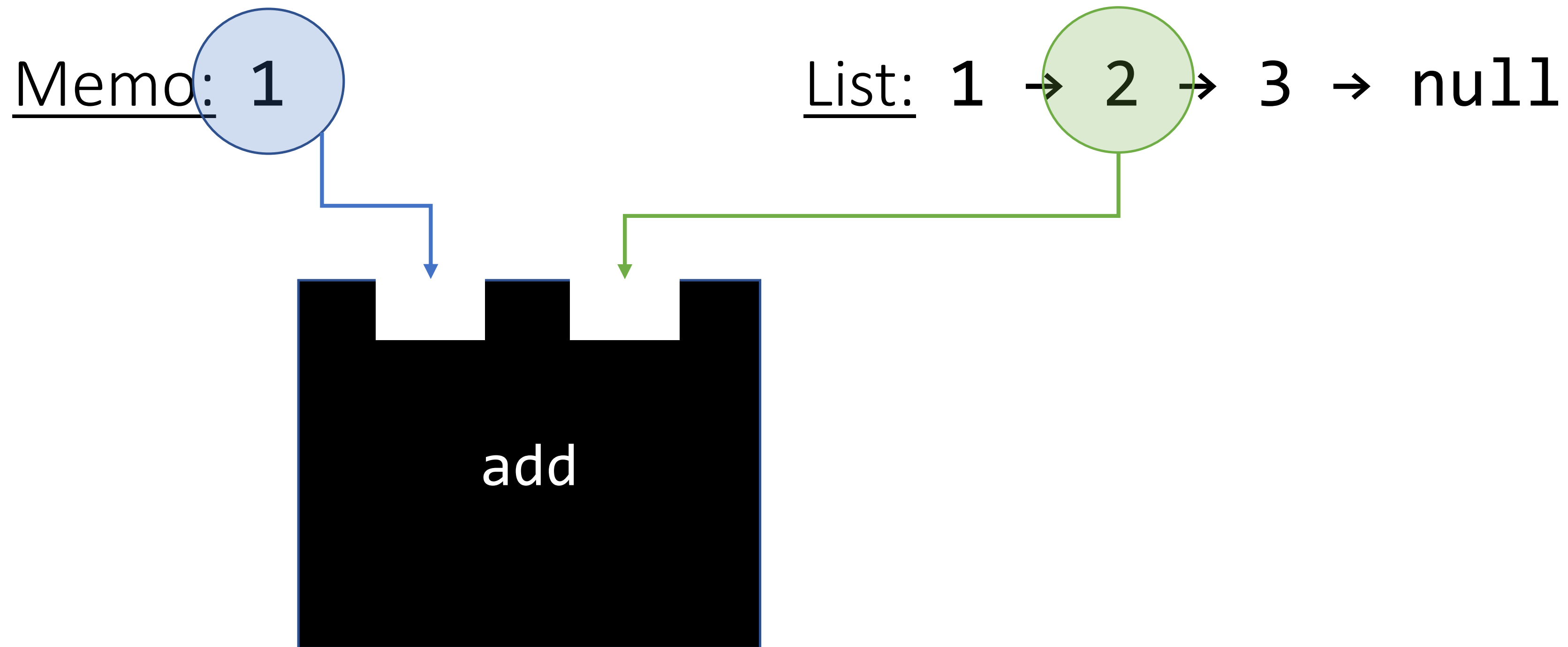
Memo: 1

List: 1 → 2 → 3 → null



This return value then becomes the next memo!

The `reduce` Function's Intuition

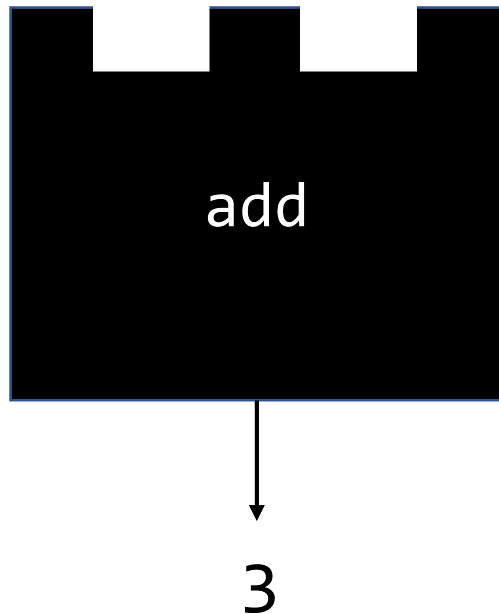


The reduce function calls add (the reducer) with memo and the next value from the list.

The `reduce` Function's Intuition

Memo: `1`

List: `1 → 2 → 3 → null`

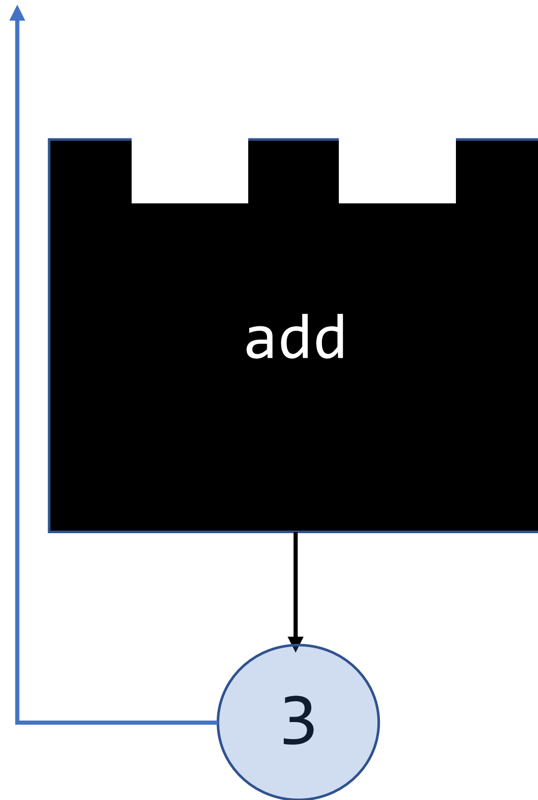


The reducer produces a return value.

The `reduce` Function's Intuition

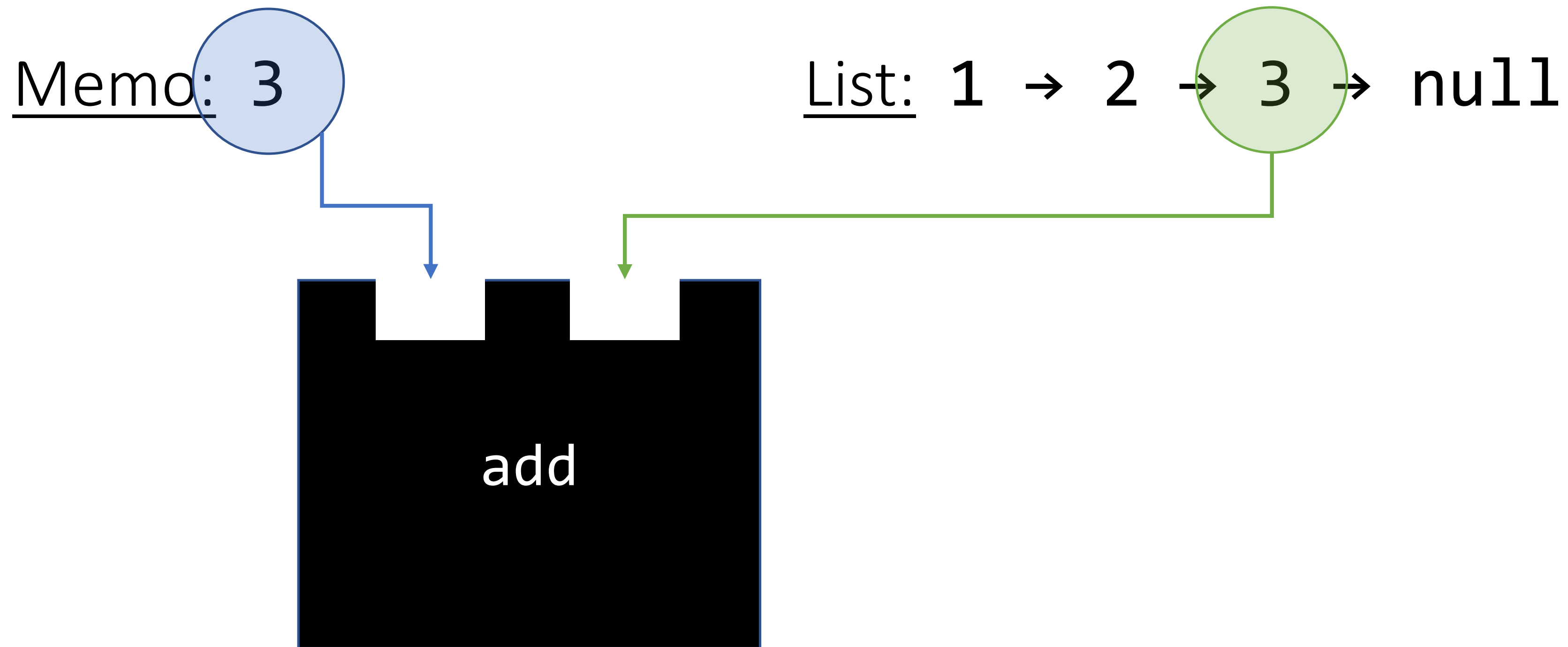
Memo: 3

List: 1 → 2 → 3 → null



This return value then becomes the next memo!

The `reduce` Function's Intuition

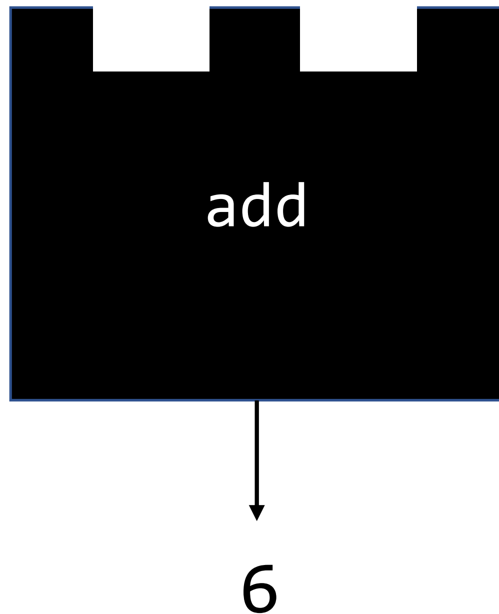


The reduce function calls add (the reducer) with memo and the next value from the list.

The `reduce` Function's Intuition

Memo: 3

List: `1 → 2 → 3 → null`

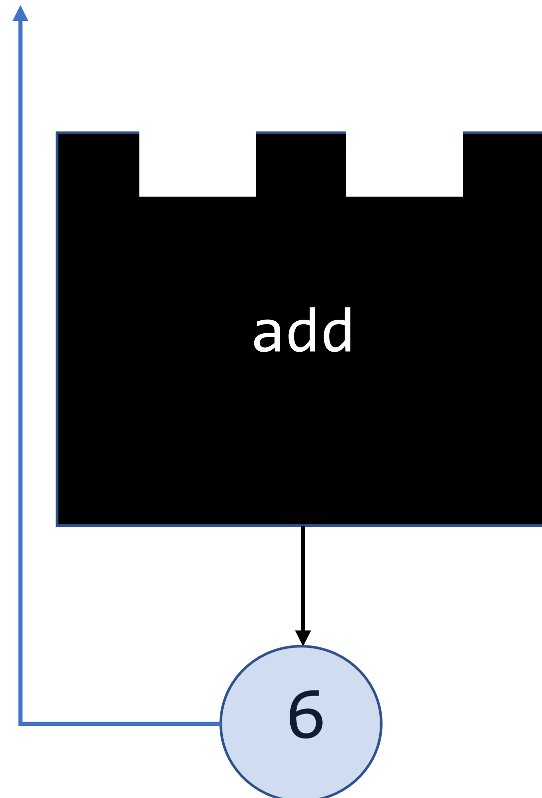


The reducer produces a return value.

The `reduce` Function's Intuition

Memo: 6

List: `1 → 2 → 3 → null`

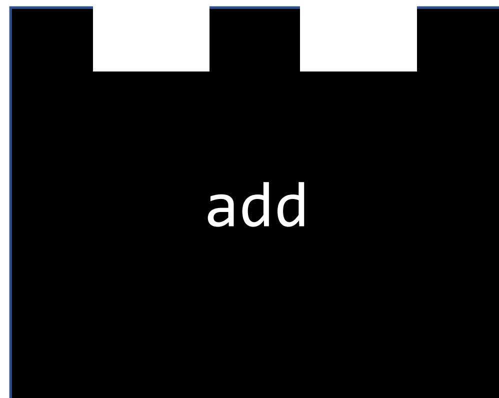


This return value then becomes the next memo!

The `reduce` Function's Intuition

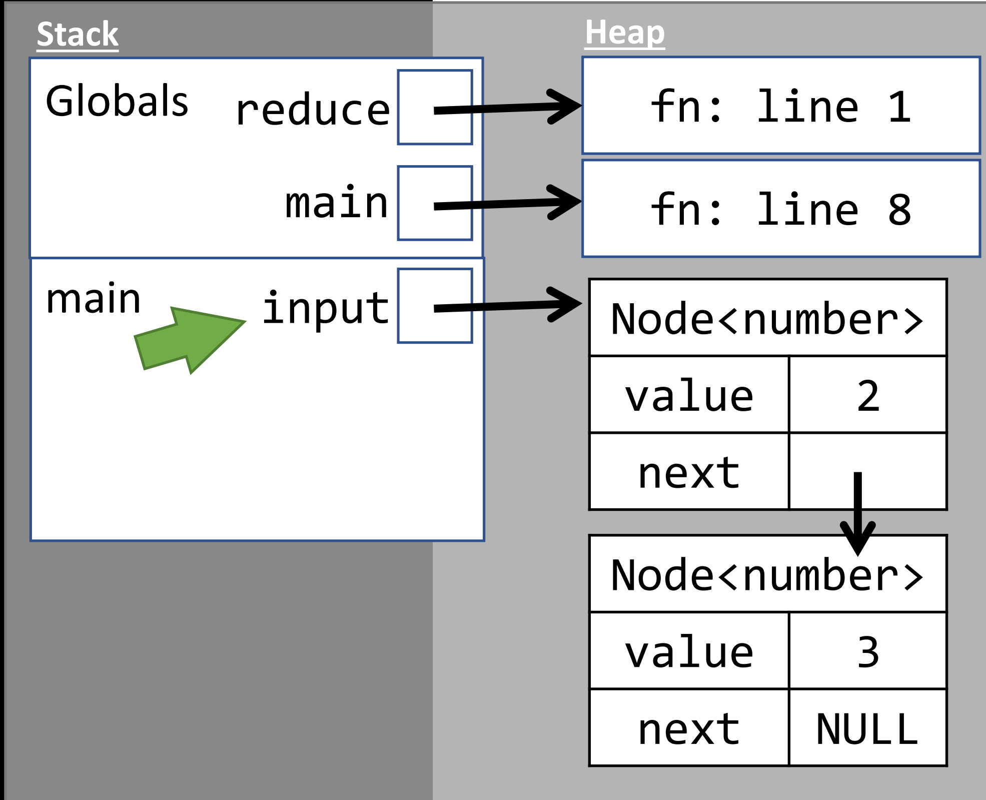
Memo: 6

List: 1 → 2 → 3 → `null`



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

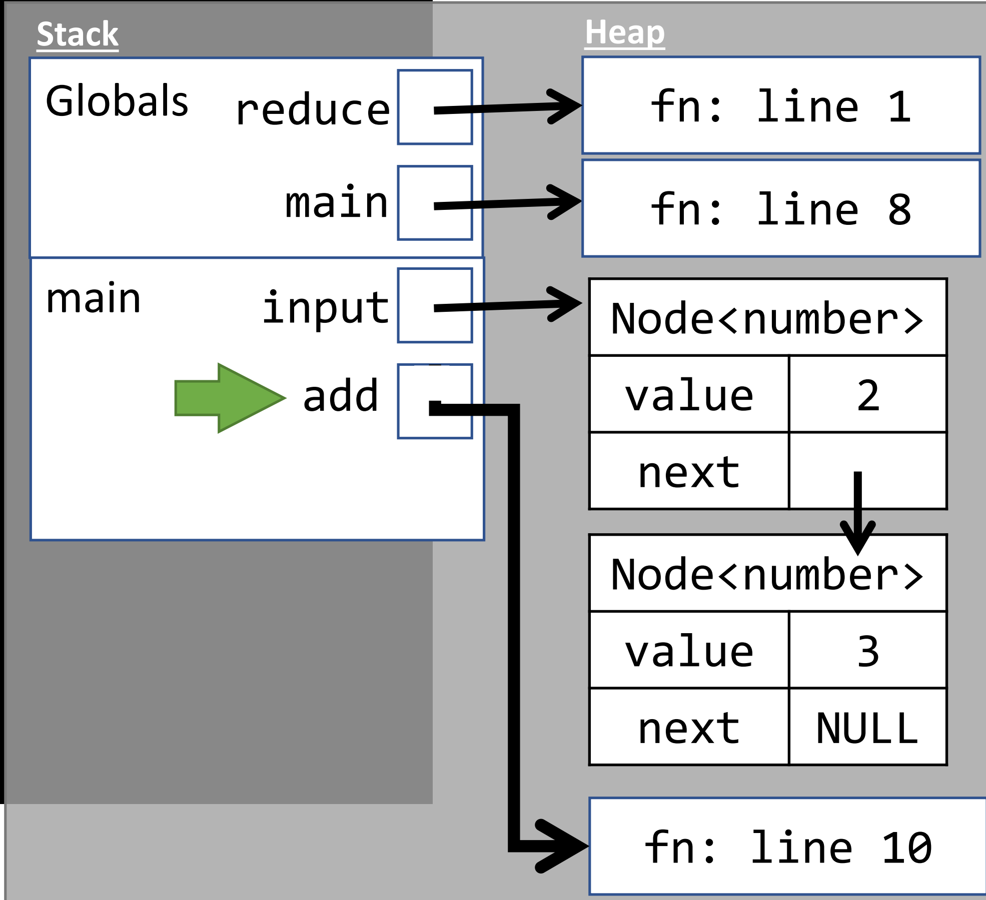
```
 1 let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
 2     if (xs === null) {
 3         return memo;
 4     } else {
 5         return reduce(rest(xs), f, f(memo, first(xs)));
 6     }
 7 };
 8 export let main = async () => {
 9     let input = listify(2, 3)
10     let add: Reducer = (memo, n) => memo + n;
11     let sum: number = reduce(input, add, 0);
12     print(sum);
13 };
14
15 main();
```



Fast-Forward

Suppose we've entered main and have initialized the input variable. Our stack/heap state will be as expected to the right.

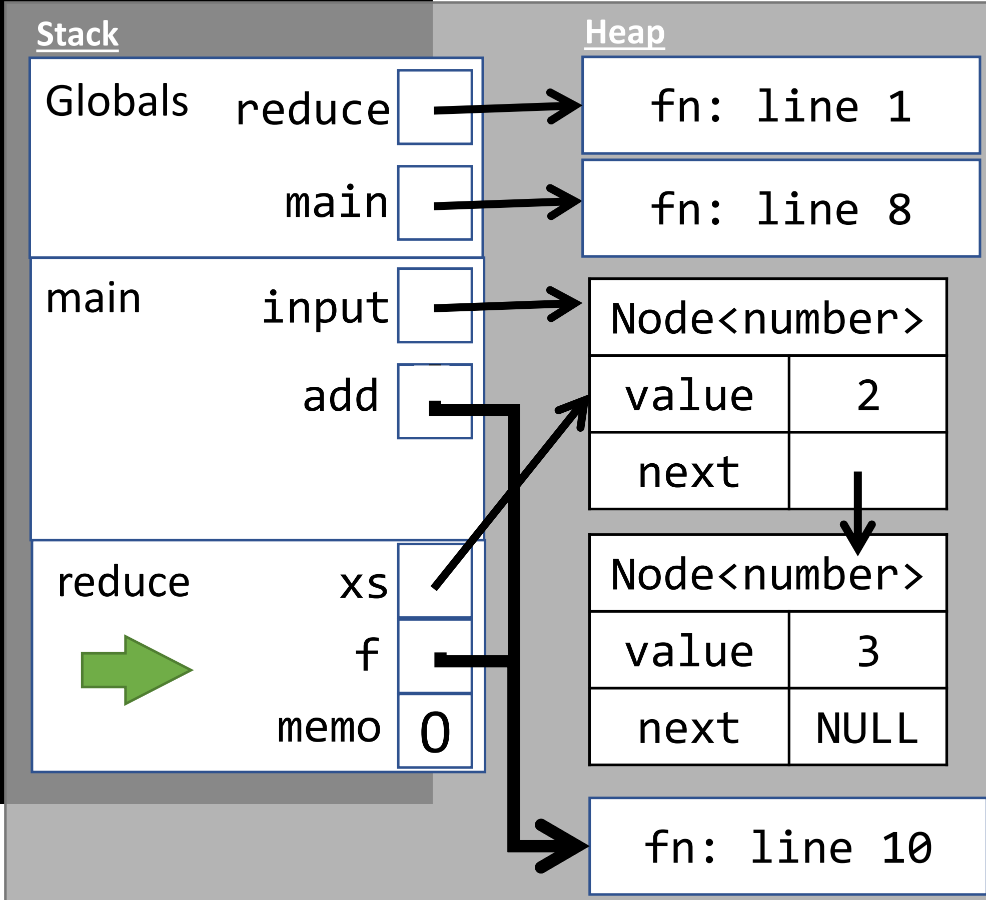
```
1  let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
2      if (xs === null) {
3          return memo;
4      } else {
5          return reduce(rest(xs), f, f(memo, first(xs)));
6      }
7  };
8  export let main = async () => {
9      let input = listify(2, 3)
10     let add: Reducer = (memo, n) => memo + n;
11     let sum: number = reduce(input, add, 0);
12     print(sum);
13 };
14
15 main();
```



Function Variable Declaration

Notice the line of a function's declaration is added to our heap notation. This is important for higher-order functions. With anonymous functions and function parameters there may be multiple names aliasing a single function definition.

```
let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
    if (xs === null) {
        return memo;
    } else {
        return reduce(rest(xs), f, f(memo, first(xs)));
    }
};
export let main = async () => {
    let input = listify(2, 3)
    let add: Reducer = (memo, n) => memo + n;
    let sum: number = reduce(input, add, 0);
    print(sum);
};

main();
```

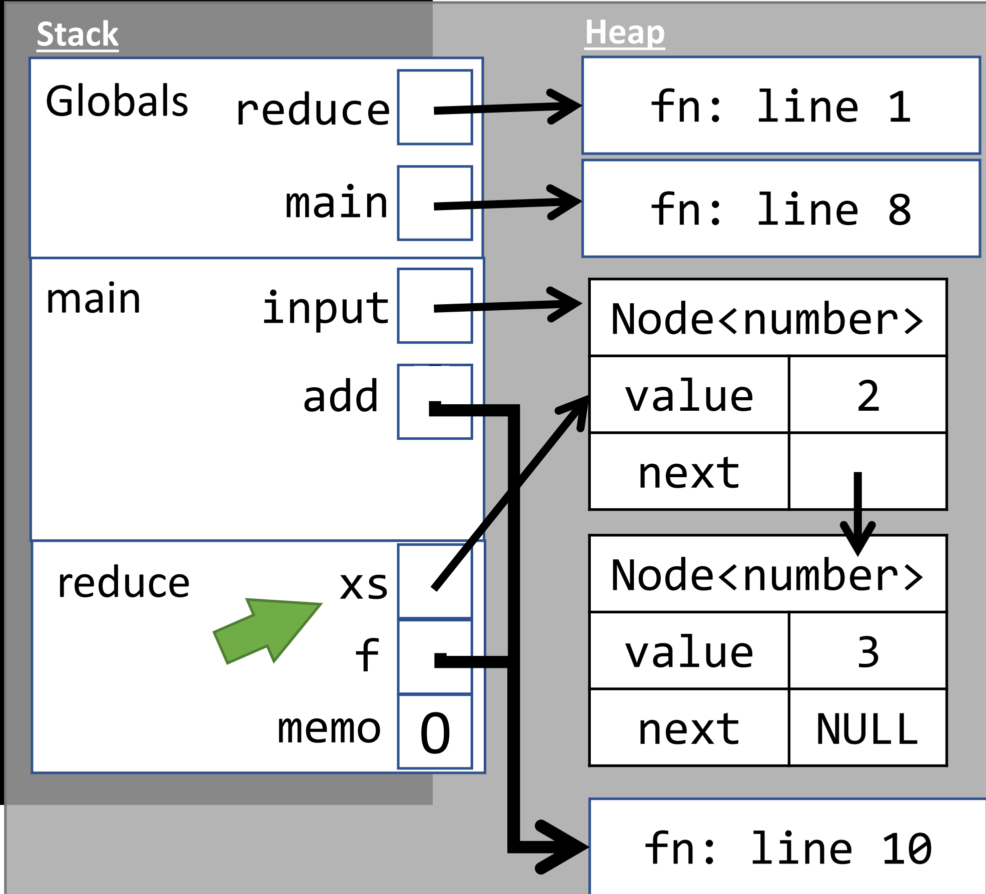


Function Call

Add frame for reduce to stack. Assign references to object and function parameters, copy value to memo.

Notice **add** in main and **f** in **reduce** refer to the same function!

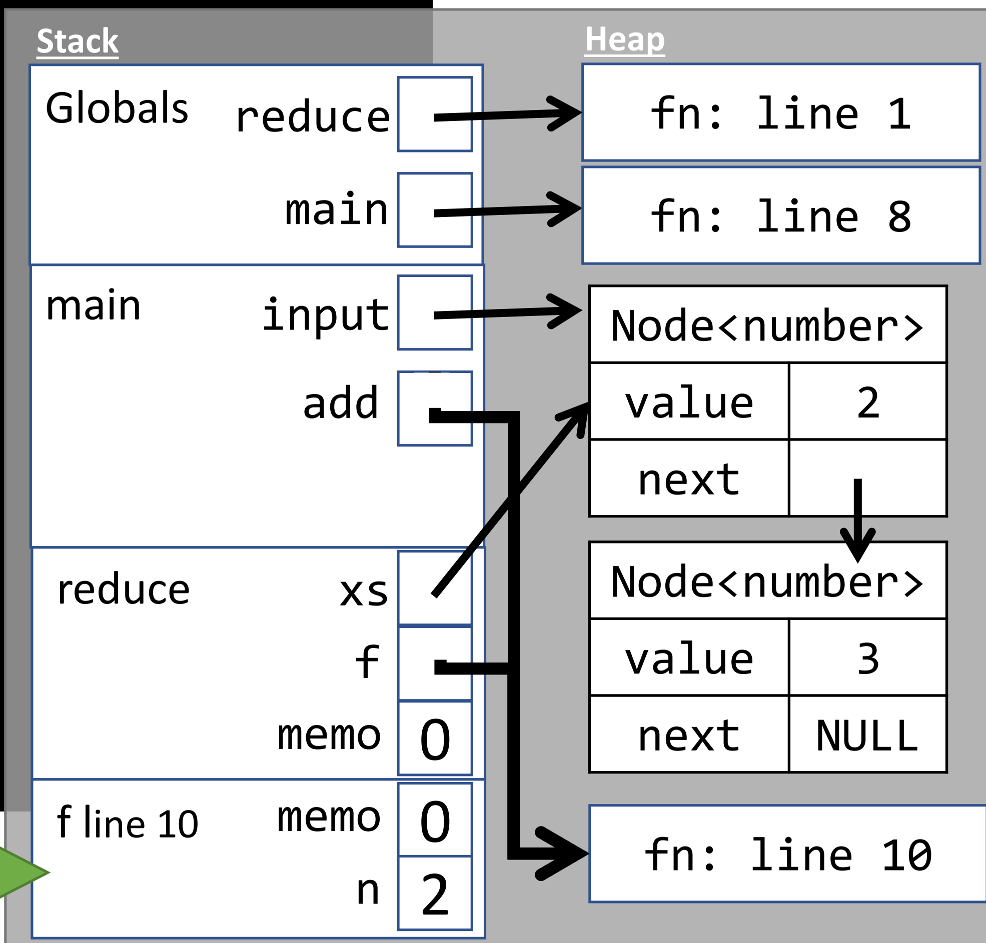
```
1  let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
2      if (xs === null) {
3          return memo;
4      } else {
5          return reduce(rest(xs), f, f(memo, first(xs)));
6      }
7  };
8  export let main = async () => {
9      let input = listify(2, 3)
10     let add: Reducer = (memo, n) => memo + n;
11     let sum: number = reduce(input, add, 0);
12     print(sum);
13 };
14
15 main();
```



Conditional + Name Resolution

The xs variable is not null, so the processor jumps to the else block.

```
1  let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
2      if (xs === null) {
3          return memo;
4      } else {
5          return reduce(rest(xs), f, f(memo, first(xs)));
6      }
7  };
8  export let main = async () => {
9      let input = listify(2, 3)
10     let add: Reducer = (memo, n) => memo + n;
11     let sum: number = reduce(input, add, 0);
12     print(sum);
13 };
14
15 main();
```

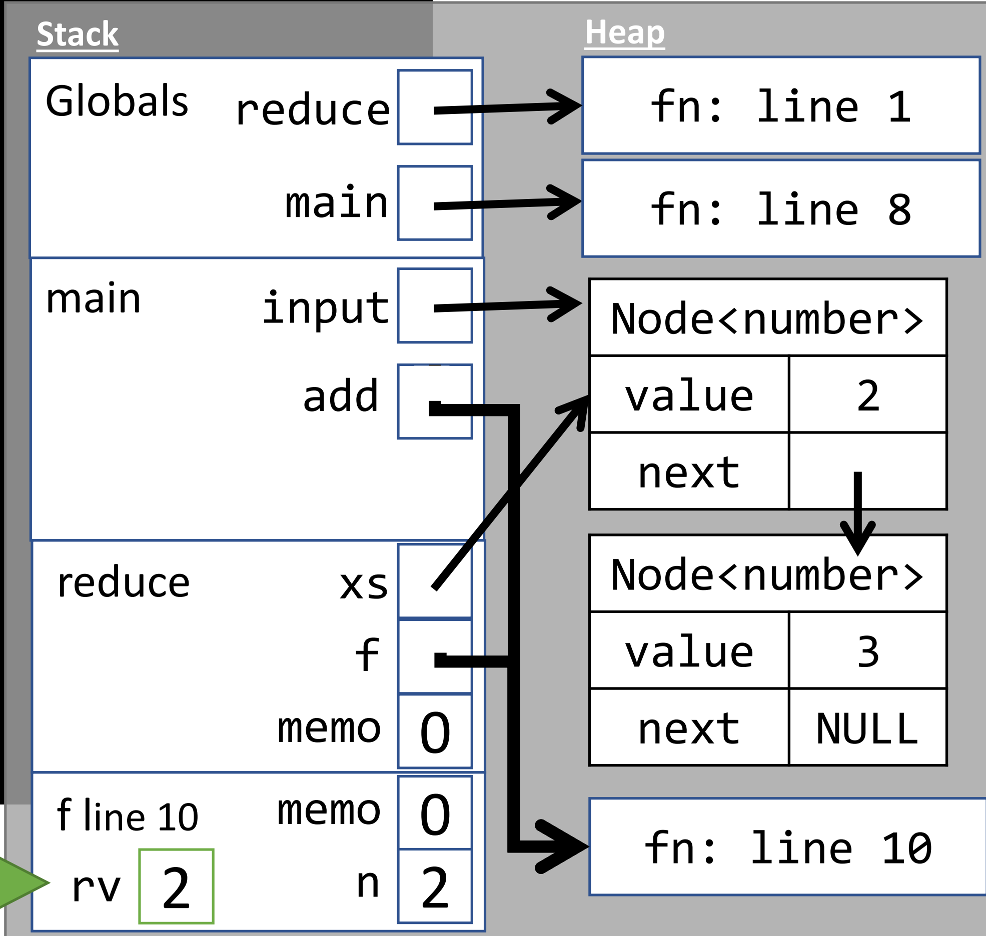


Function Call to Function Parameter

The processor needs to evaluate `f(memo, first(xs))`

When using name resolution to find function **f**, notice it is defined in the current frame and not in Globals. When this is the case, When establishing the new frame, you should add its line number.

```
1  let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
2      if (xs === null) {
3          return memo;
4      } else {
5          return reduce(rest(xs), f, f(memo, first(xs)));
6      }
7  };
8  export let main = async () => {
9      let input = listify(2, 3)
10     let add: Reducer = (memo, n) => memo + n;
11     let sum: number = reduce(input, add, 0);
12     print(sum);
13 };
14
15 main();
```

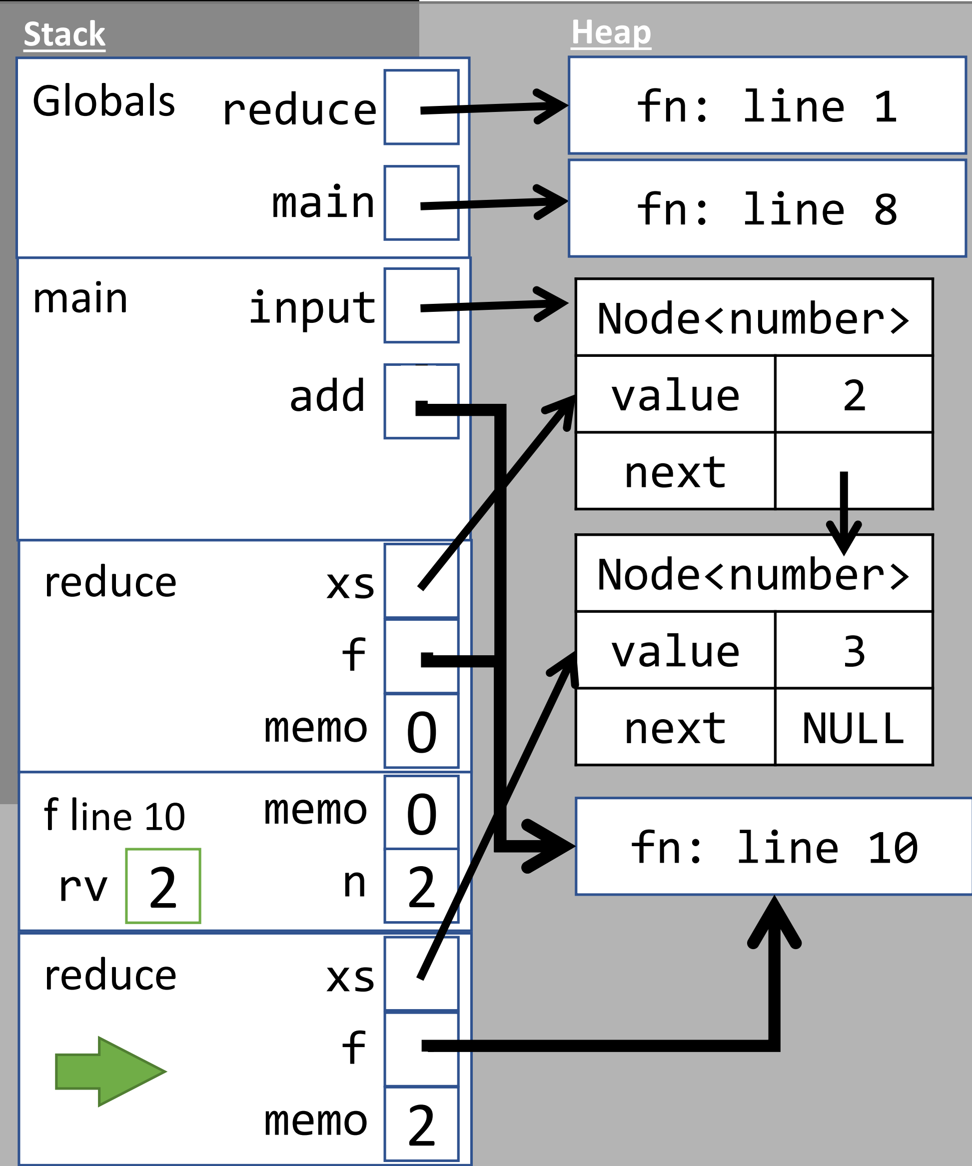


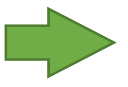
Return Statement

Notice this function simply adds **memo** and **n** and returns the result. The return is implicit thanks to the shorthand function syntax.

Rather than erasing the frame, which the processor *does* do, we will write in a return value field, "**rv**", with the value returned. This helps avoid erasing on paper and shows the complete trail.

```
1  let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
2      if (xs === null) {
3          return memo;
4      } else {
5          return reduce(rest(xs), f, f(memo, first(xs)));
6      }
7  };
8  export let main = async () => {
9      let input = listify(2, 3)
10     let add: Reducer = (memo, n) => memo + n;
11     let sum: number = reduce(input, add, 0);
12     print(sum);
13 };
14
15 main();
```

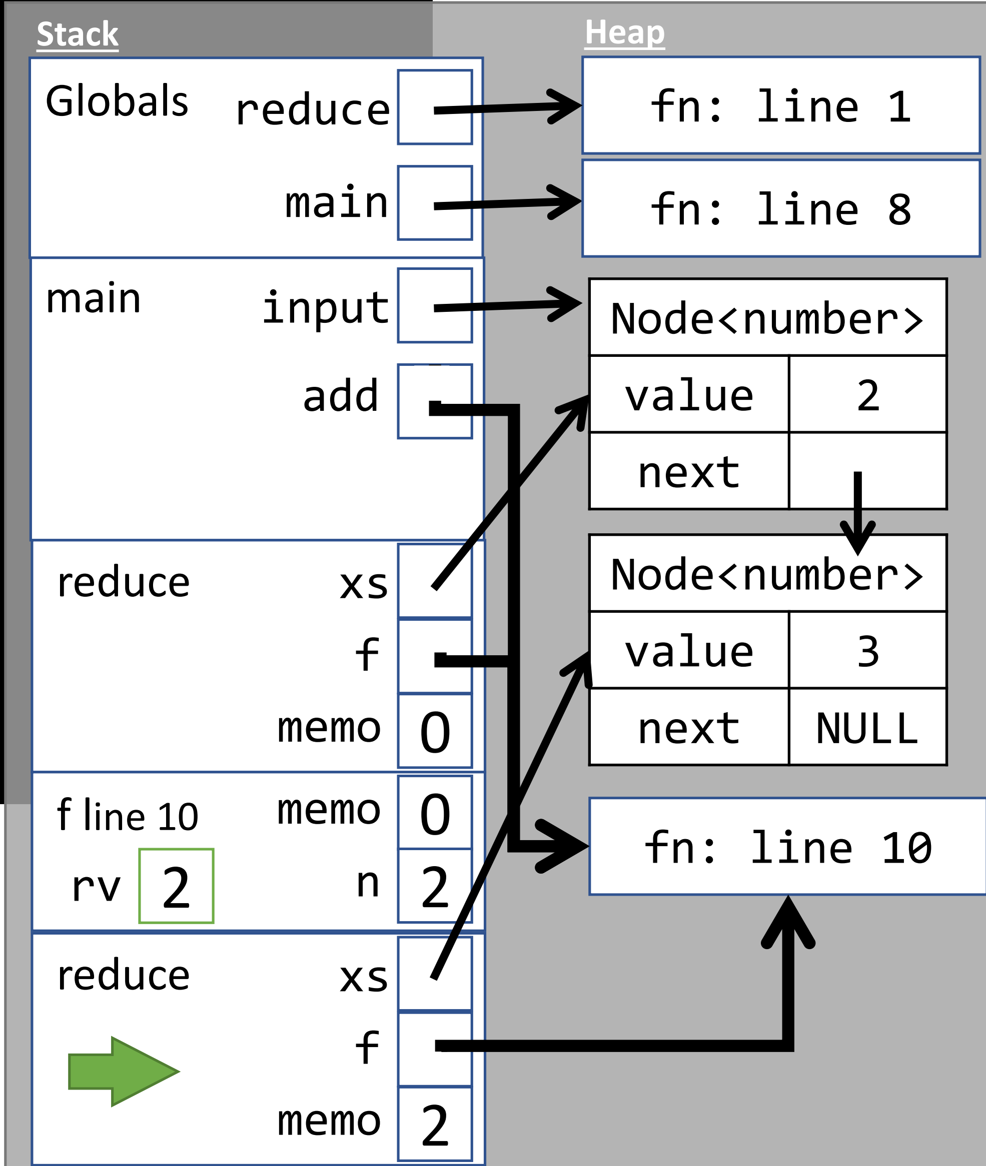




Recursive Call

Notice we're adding another reduce frame. This time, though, the **memo** is **2** which was the return value of calling **f**!

```
1  let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
2      if (xs === null) {
3          return memo;
4      } else {
5          return reduce(rest(xs), f, f(memo, first(xs)));
6      }
7  };
8  export let main = async () => {
9      let input = listify(2, 3)
10     let add: Reducer = (memo, n) => memo + n;
11     let sum: number = reduce(input, add, 0);
12     print(sum);
13 };
14
15 main();
```



Conditional + Name Resolution

The xs variable is not null, so the processor jumps to the else block.

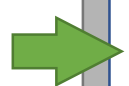
```
1  let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
2      if (xs === null) {
3          return memo;
4      } else {
5          return reduce(rest(xs), f, f(memo, first(xs)));
6      }
7  };
8  export let main = async () => {
9      let input = listify(2, 3)
10     let add: Reducer = (memo, n) => memo + n;
11     let sum: number = reduce(input, add, 0);
12     print(sum);
13 };
14
15 main();
```

Stack			Heap
Globals	reduce	→	fn: line 1
	main	→	fn: line 8
main	input	→	Node<number> (value 2, next →)
	add	→	fn: line 10
reduce	xs	→	Node<number> (value 2)
	f	→	fn: line 10
	memo	0	
f line 10	memo	0	
	n	2	
reduce	xs	→	Node<number> (value 3, next NULL)
	f	→	fn: line 10
	memo	2	
f line 10	memo	2	
	n	3	

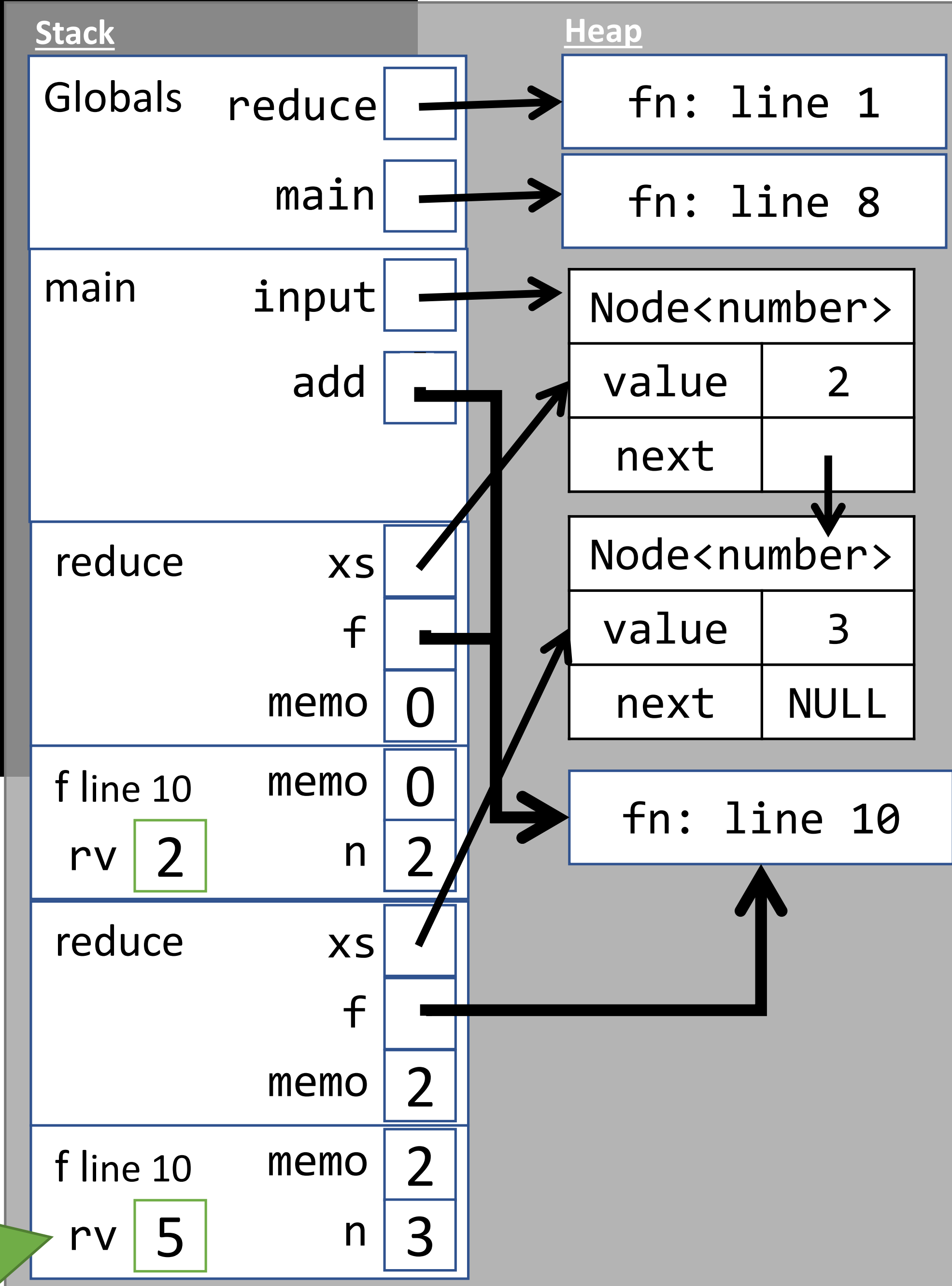
Function Call to Function Parameter

The processor needs to evaluate `f(memo, first(xs))`

When using name resolution to find function **f**, notice it is defined in the current frame and not in Globals. When this is the case, When establishing the new frame, you should add its line number.



```
1  let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
2      if (xs === null) {
3          return memo;
4      } else {
5          return reduce(rest(xs), f, f(memo, first(xs)));
6      }
7  };
8  export let main = async () => {
9      let input = listify(2, 3)
10     let add: Reducer = (memo, n) => memo + n;
11     let sum: number = reduce(input, add, 0);
12     print(sum);
13 };
14
15 main();
```



Return Statement

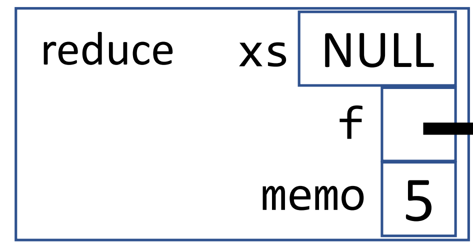
Notice this function simply adds **memo** and **n** and returns the result. The return is implicit thanks to the shorthand function syntax.

Rather than erasing the frame, which the processor *does* do, we will write in a return value field, "**rv**", with the value returned. This helps avoid erasing on paper and shows the complete trail.

```typescript
let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
    if (xs === null) {
        return memo;
    } else {
        return reduce(rest(xs), f, f(memo, first(xs)));
    }
};
export let main = async () => {
    let input = listify(2, 3)
    let add: Reducer = (memo, n) => memo + n;
    let sum: number = reduce(input, add, 0);
    print(sum);
};

main();
```

Recursive Call

Assume the reduce frame to the right is below the others on the stack. Notice the next memo is the return value of the previous reducer.



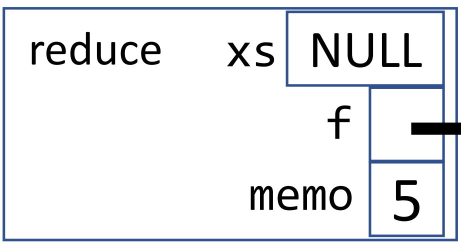
Stack
Globals reduce
main
main input
add
reduce xs
f
memo 0
f line 10 memo 0
rv 2 n 2
reduce xs
f
memo 2
f line 10 memo 2
rv 5 n 3

Heap
fn: line 1
fn: line 8
Node<number>
value 2
next
Node<number>
value 3
next NULL
fn: line 10

```
1  let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
2      if (xs === null) {
3          return memo;
4      } else {
5          return reduce(rest(xs), f, f(memo, first(xs)));
6      }
7  };
8  export let main = async () => {
9      let input = listify(2, 3)
10     let add: Reducer = (memo, n) => memo + n;
11     let sum: number = reduce(input, add, 0);
12     print(sum);
13 };
14
15 main();
```

Conditional + Name Resolution

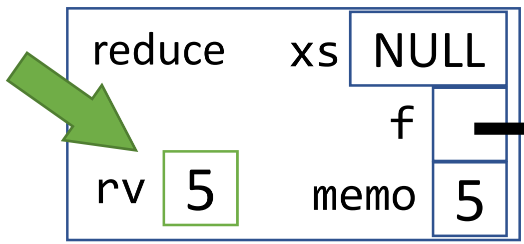
The xs variable IS null! So the processor enters the then block...



```
1  let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
2      if (xs === null) {
3          return memo;
4      } else {
5          return reduce(rest(xs), f, f(memo, first(xs)));
6      }
7  };
8  export let main = async () => {
9      let input = listify(2, 3)
10     let add: Reducer = (memo, n) => memo + n;
11     let sum: number = reduce(input, add, 0);
12     print(sum);
13 };
14
15 main();
```


Return Statement

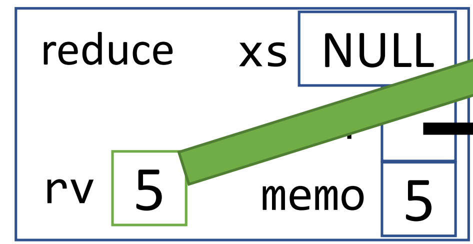
The returned value is simply memo, so we add the return value to the frame and then return back to the last function which hasn't returned.



```
1  let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
2      if (xs === null) {
3          return memo;
4      } else {
5          return reduce(rest(xs), f, f(memo, first(xs)));
6      }
7  };
8  export let main = async () => {
9      let input = listify(2, 3)
10     let add: Reducer = (memo, n) => memo + n;
11     let sum: number = reduce(input, add, 0);
12     print(sum);
13 };
14
15 main();
```

Return Statement

Notice the recursive call itself is the value being returned. So we fill in its return value and then return it to the recursive call it originated from.



Stack

Globals reduce main

main input add

reduce xs f memo 0

f line 10 memo 0 rv 2 n 2

reduce xs f rv 5 memo 2

f line 10 memo 2 rv 5 n 3

Heap

fn: line 1

fn: line 8

Node<number>	
value	2
next	

Node<number>	
value	3
next	NULL

fn: line 10

```
1  let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
2      if (xs === null) {
3          return memo;
4      } else {
5          return reduce(rest(xs), f, f(memo, first(xs)));
6      }
7  };
8  export let main = async () => {
9      let input = listify(2, 3)
10     let add: Reducer = (memo, n) => memo + n;
11     let sum: number = reduce(input, add, 0);
12     print(sum);
13 };
14
15 main();
```

Stack

Globals: reduce, main

main: input, add

reduce: xs, f, memo 0, rv 5

f line 10: memo 0, n 2, rv

reduce: xs, f, memo 2, rv 5

f line 10: memo 2, n 3, rv 5

reduce: xs NULL, f, memo 5, rv 5

Heap

fn: line 1

fn: line 8

Node<number>: value 2, next

Node<number>: value 3, next NULL

fn: line 10

Return Statement

Notice the recursive call itself is the value being returned. So we fill in its return value and then return it to the recursive call it originated from.

```
1  let reduce = (xs: Node<number>, f: Reducer, memo: number): number => {
2      if (xs === null) {
3          return memo;
4      } else {
5          return reduce(rest(xs), f, f(memo, first(xs)));
6      }
7  };
8  export let main = async () => {
9      let input = listify(2, 3)
10     let add: Reducer = (memo, n) => memo + n;
11     let sum: number = reduce(input, add, 0);
12     print(sum);
13 };
14
15 main();
```

Stack

Globals reduce main

main input add sum 5

reduce xs f rv 5 memo 0

f line 10 memo 0 rv 2 n 2

reduce xs f rv 5 memo 2

f line 10 memo 2 rv 5 n 3

Heap

fn: line 1

fn: line 8

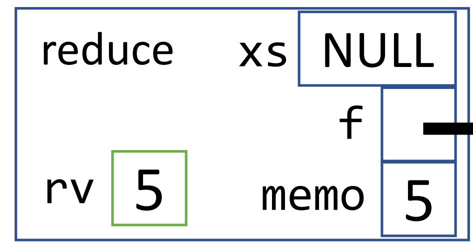
Node<number> value 2 next

Node<number> value 3 next NULL

fn: line 10

Return Statement

Notice the recursive call itself is the value being returned. So we fill in its return value and the original reduce frame returns its value back to the main function to initialize sum.



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

- Open lec16 / 02-reduce-app.ts
- Goal: After loading Berry's game data from data/berry-stats-2018.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);
```

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

List<Game>

Filter →

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

List<Game>

Map →

20
13

List<number>

Reduce →

33

number

O

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: List<Game> ⟶ List<Game>
Map:    List<Game> → List<number>
Reduce: List<number> → number
```

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