`map`, **`reduce`**, and Type Inference

Lecture 13 – COMP110 – Spring 2018

Announcements

- Review Session: Tomorrow at 5pm in SN14
  - If you were dissatisfied with your midterm grade, you should come!
- Office hours close Friday at 12pm for Spring Break
  - If you anticipate needing to ask questions about PS03 you should do so before Friday afternoon
- No tutoring this week because of the early Friday close
  - Come to office hours with conceptual help questions

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
```

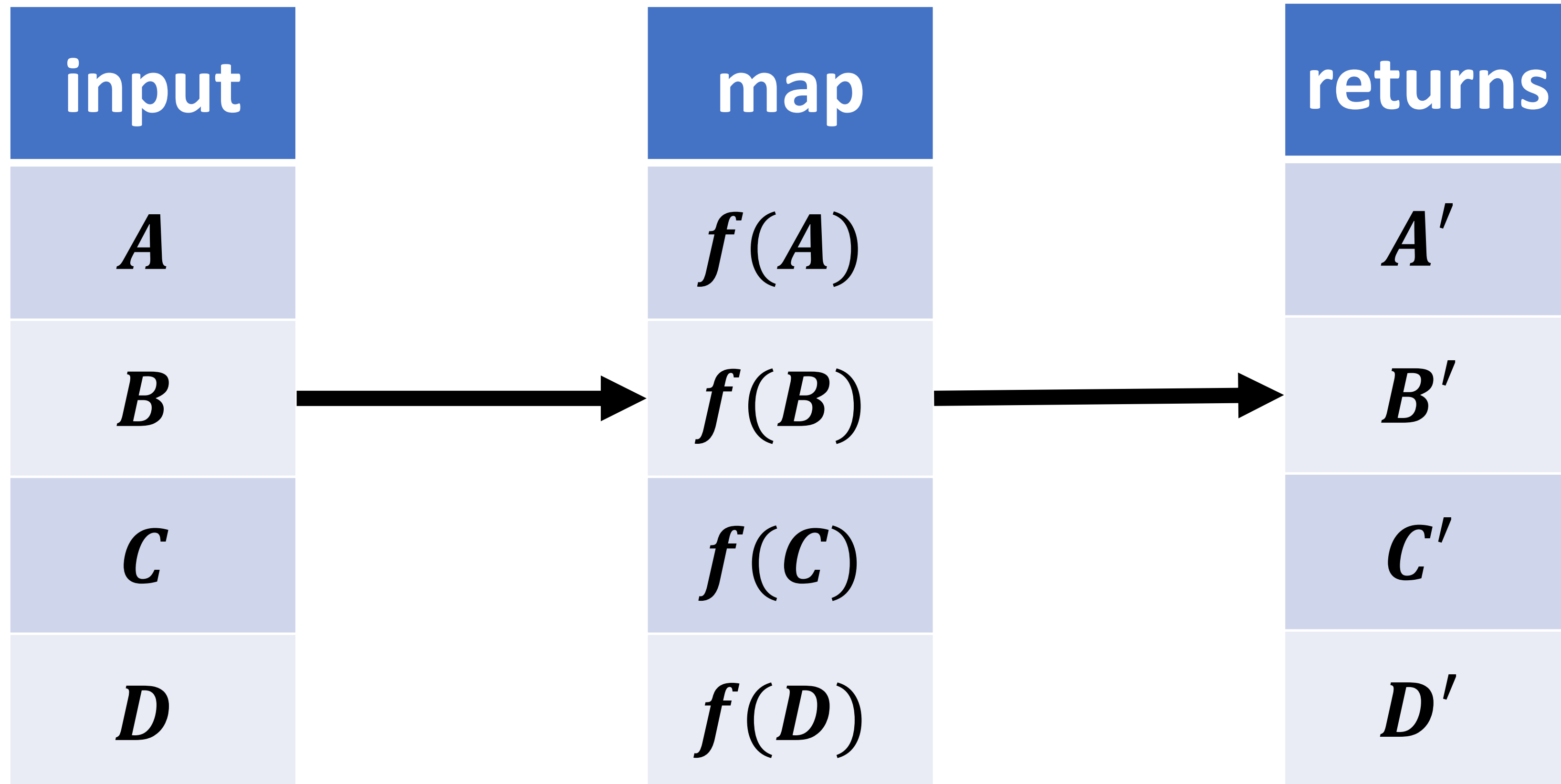
2. Given the definitions on the left, after the code below completes, what is the result variable's type and what is its value?

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

let map = <T, U> (xs: List<T>, f: Transform<T, U>): List<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: List<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!*

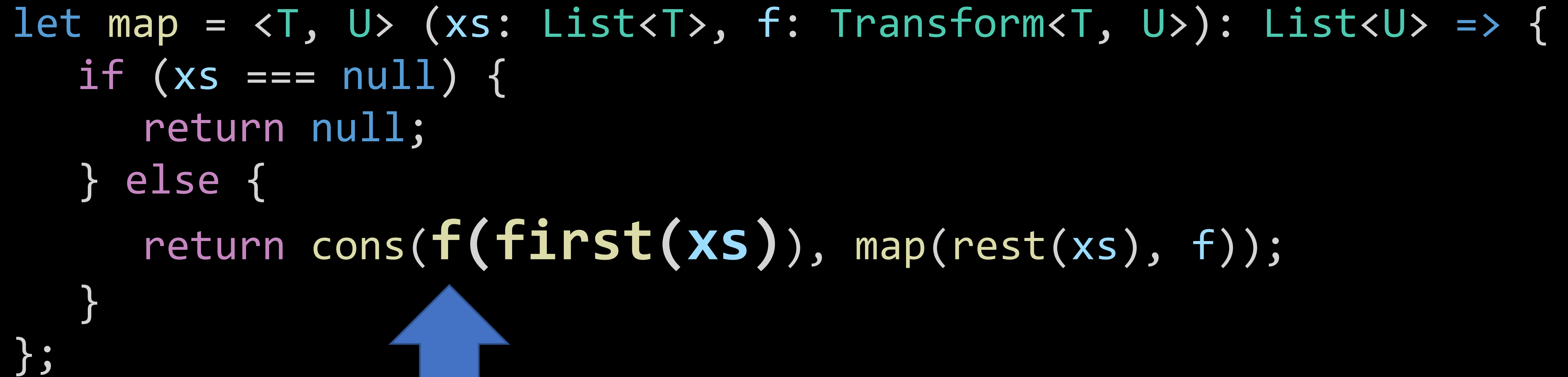
1

```
(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 lec13 / 00-map-app.ts
- Goal: After loading Berry's game data from data/berry-stats-2018.csv, produce a list of *only* Berry's points per game values

1. **TODO #1)** Declare a (Transform) 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)** In the main function, **assign** to the variable ***points*** the result of mapping the ***gamesToPoints*** function over the ***games*** list.
   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);
```

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: List<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?

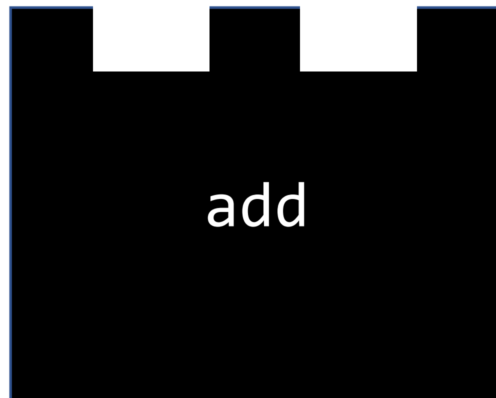
```
let add = (memo: number, item: number): number => {
    return memo + item;
};
```

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

The `reduce` Function's Intuition

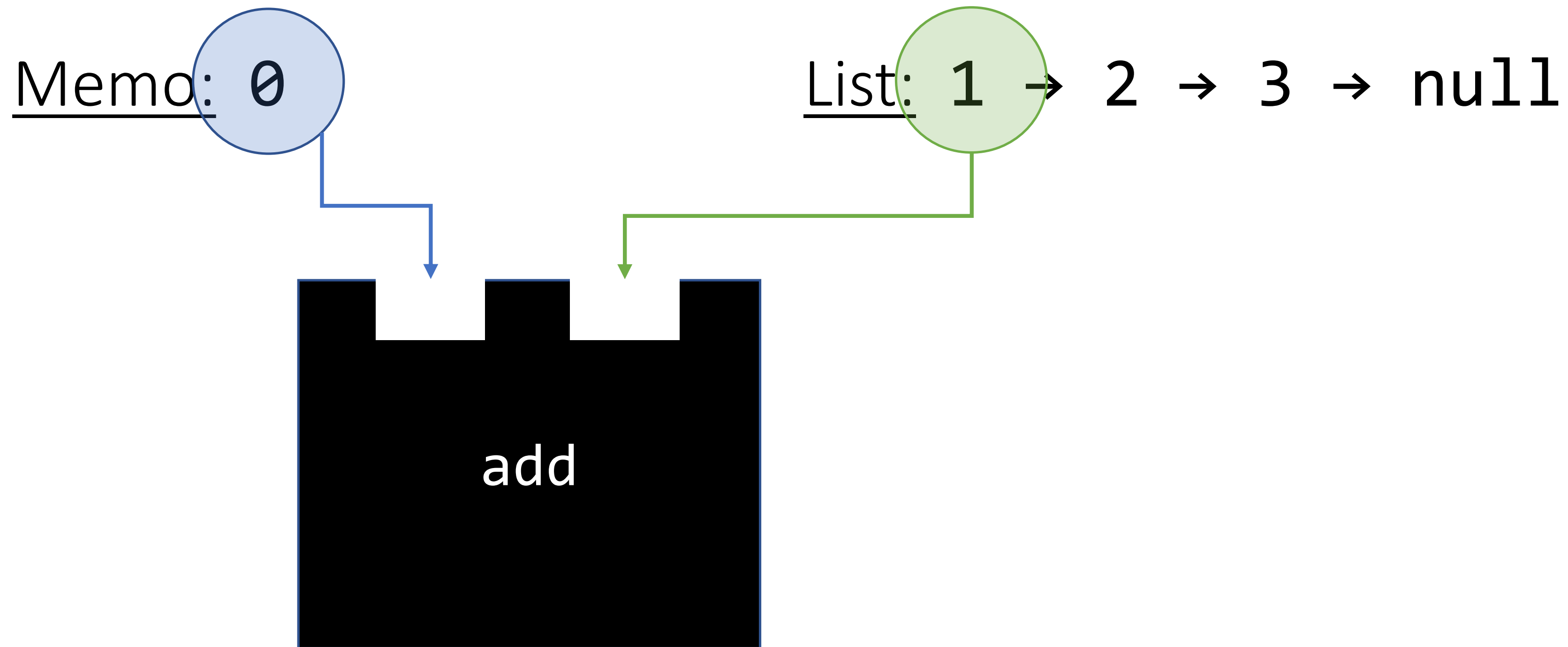
Memo: `0`

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

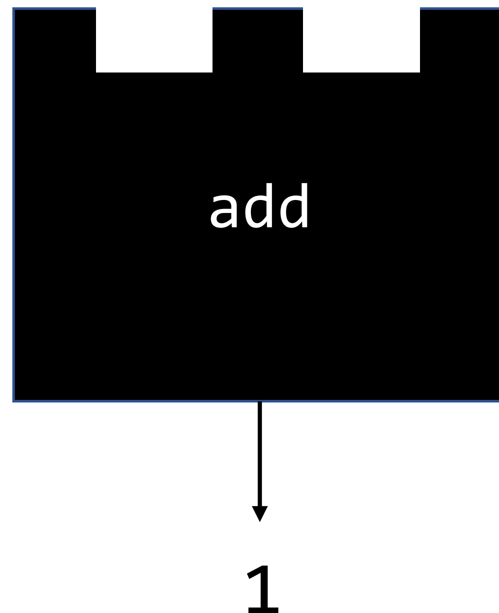


The reduce function calls add (the reducer) with memo and the next value from the list.

The `reduce` Function's Intuition

Memo: `0`

List: `1 → 2 → 3 → null`

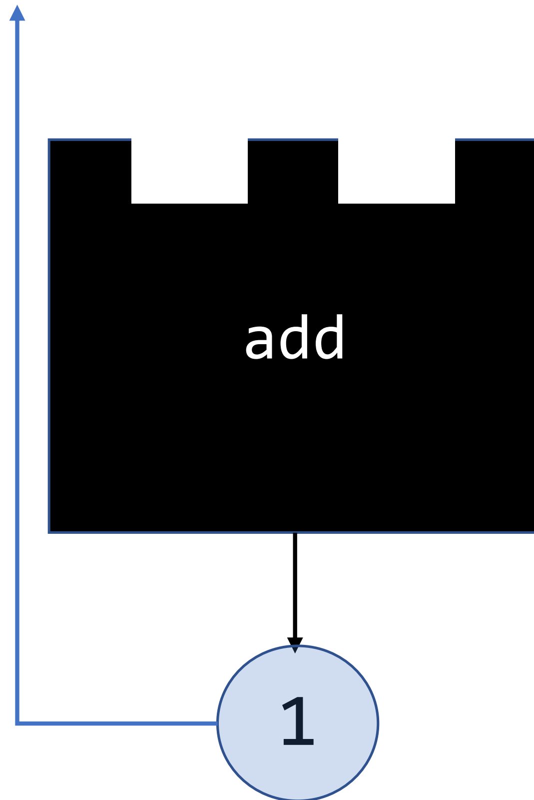


The reducer produces a return value.

The `reduce` Function's Intuition

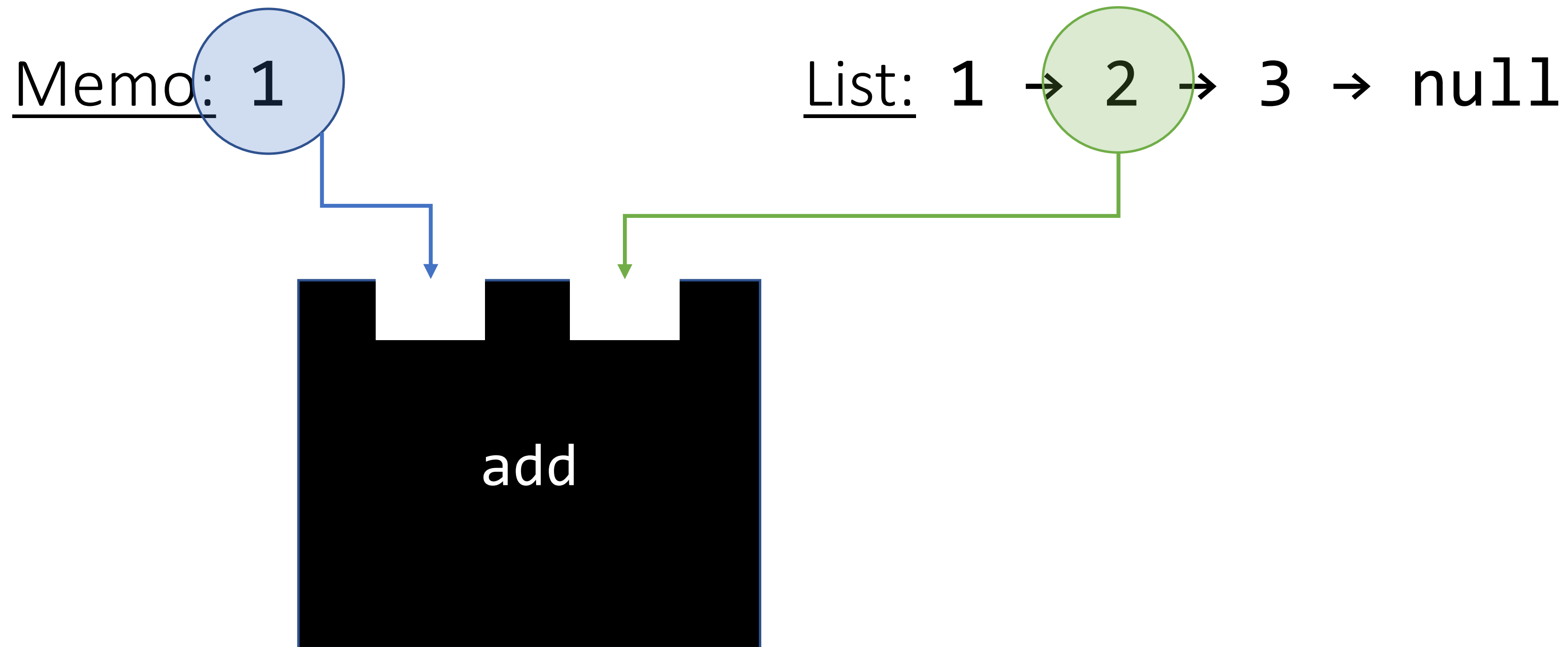
Memo: 1

List: 1 → 2 → 3 → null



This return value then becomes the next memo!

The `reduce` Function's Intuition

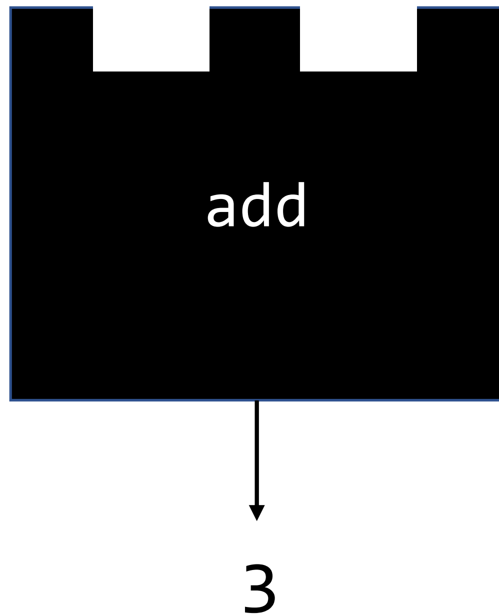


The reduce function calls add (the reducer) with memo and the next value from the list.

The `reduce` Function's Intuition

Memo: 1

List: 1 → 2 → 3 → null

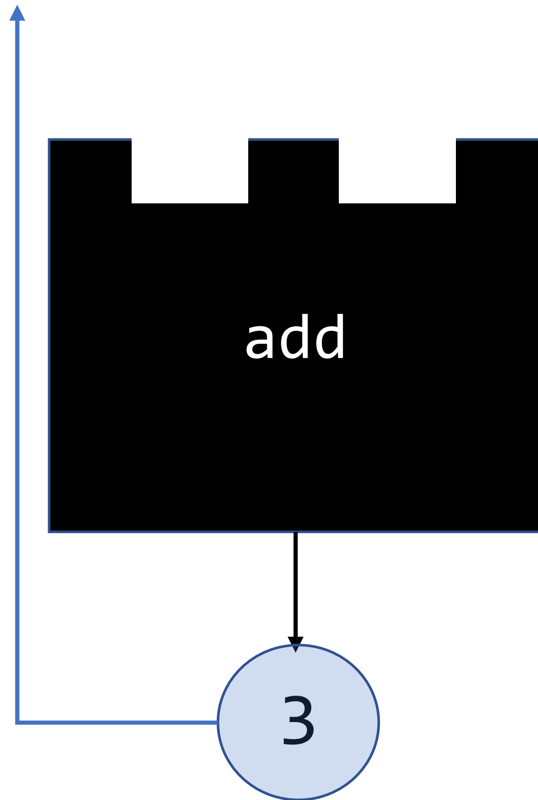


The reducer produces a return value.

The `reduce` Function's Intuition

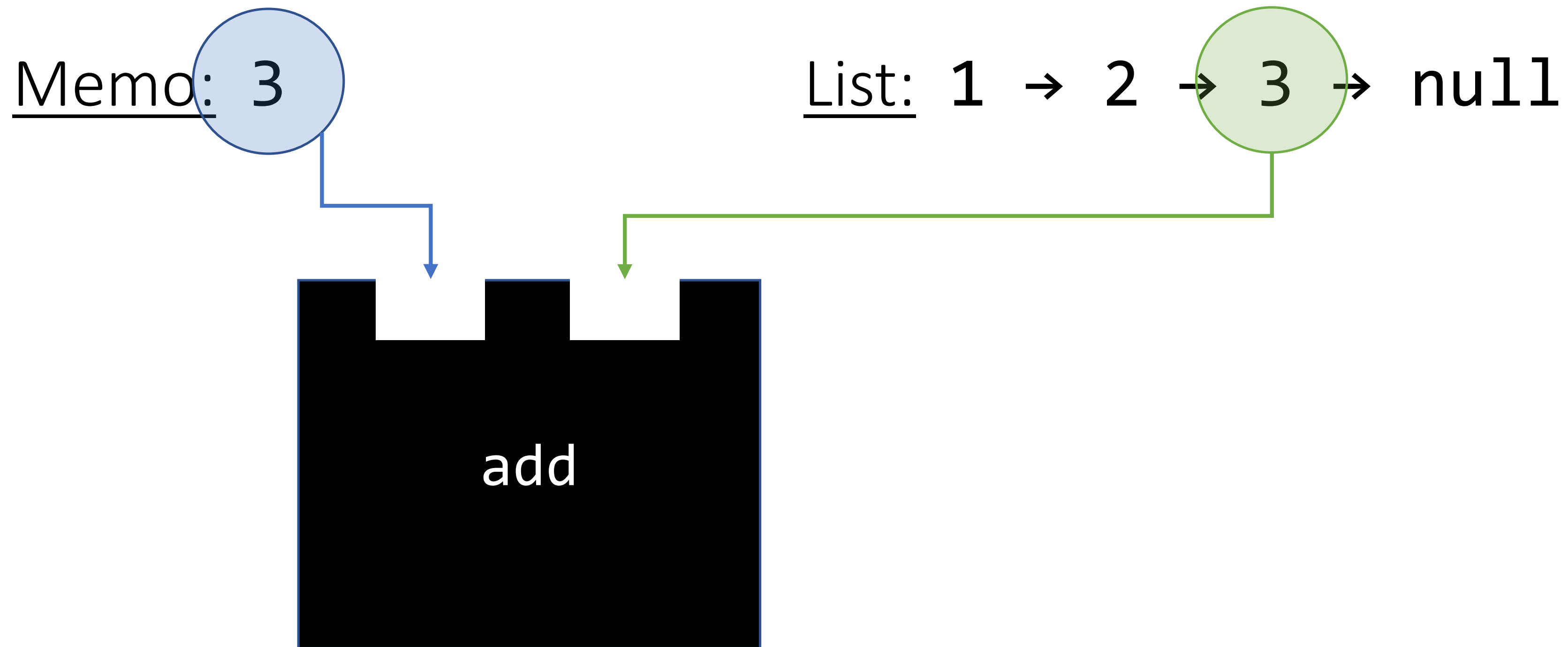
Memo: 3

List: 1 → 2 → 3 → null



This return value then becomes the next memo!

The `reduce` Function's Intuition

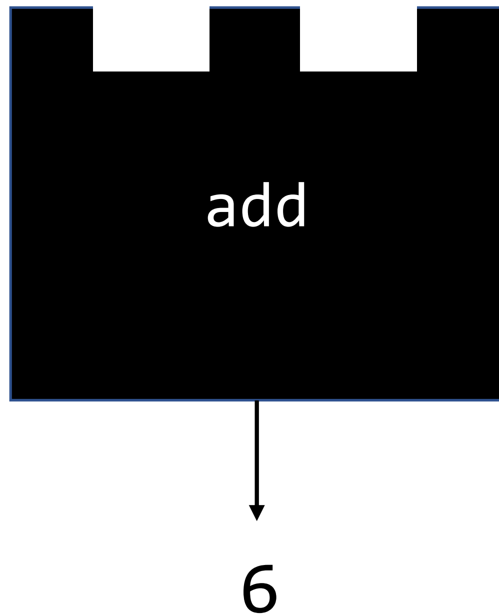


The reduce function calls add (the reducer) with memo and the next value from the list.

The `reduce` Function's Intuition

Memo: 3

List: `1 → 2 → 3 → null`

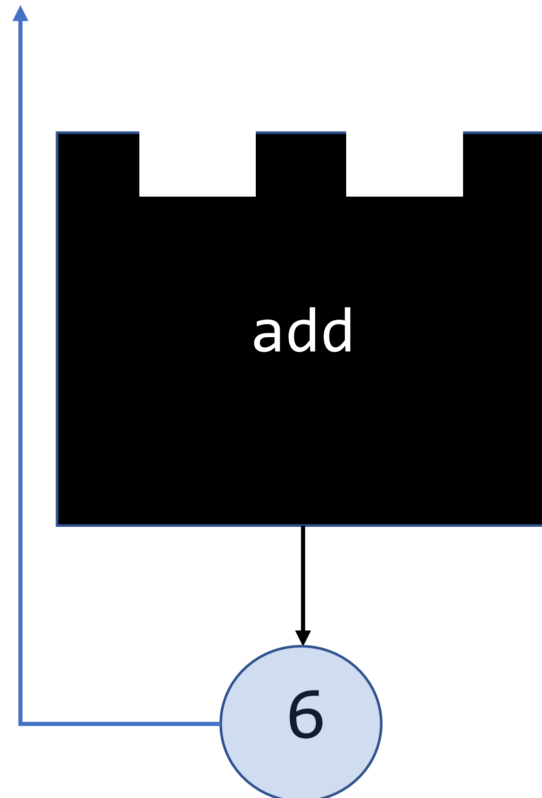


The reducer produces a return value.

The `reduce` Function's Intuition

Memo: 6

List: `1 → 2 → 3 → null`

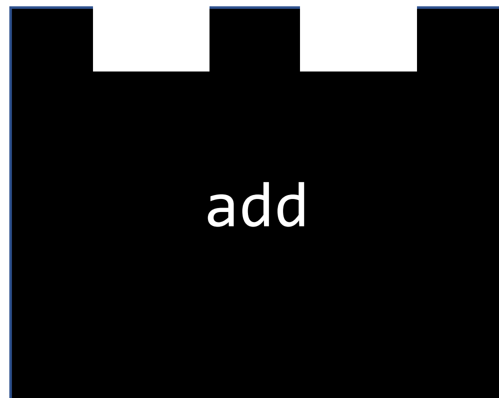


This return value then becomes the next memo!

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**.

Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

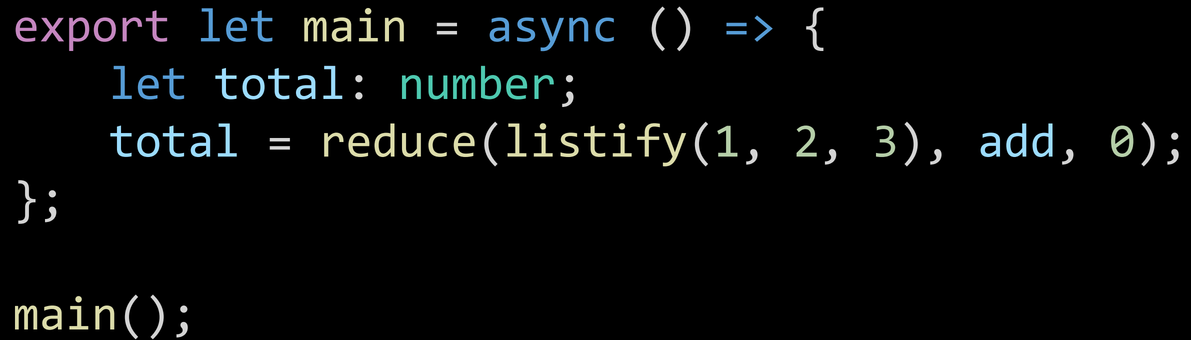
In the following slides we will trace through this program line-by-line.

*Note that to fit the **reduce** function on this slide it is not the fully generic form of reduce we'll otherwise use.*

Tracing **reduce**

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

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

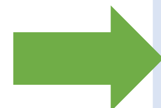


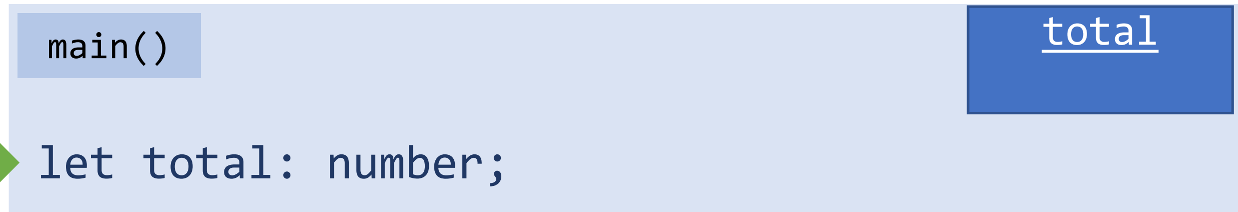
Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

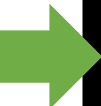




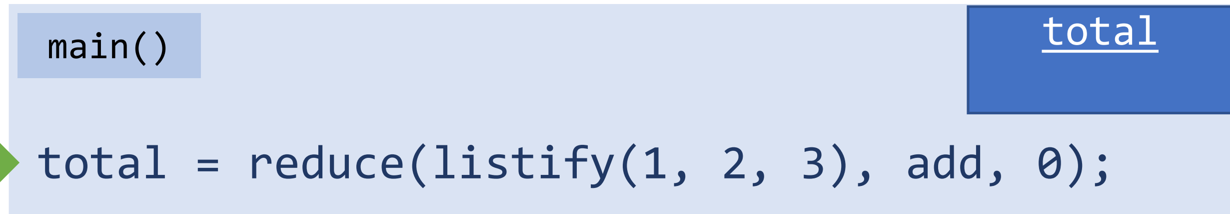
Tracing **reduce**

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

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



```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```



Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

reduce(1→2→3→null, add, 0)

xs	f	memo
1→2→3→null	add	0

if (xs === null)

main()

total

total = reduce(listify(1, 2, 3), add, 0);

Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

reduce(1→2→3→null, add, 0)

xs	f	memo
1→2→3→null	add	0

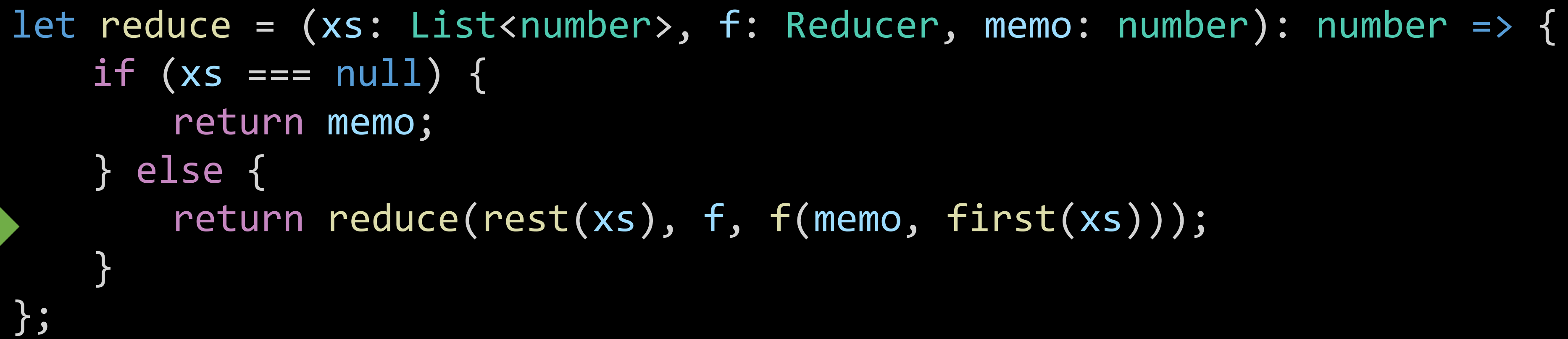
if (false)

main()

total

total = reduce(listify(1, 2, 3), add, 0);

Tracing **reduce**



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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

reduce(1→2→3→null, add, 0)

xs	f	memo
1→2→3→null	add	0

```
return reduce(rest(xs), f, f(memo, first(xs)));
```

main()

total

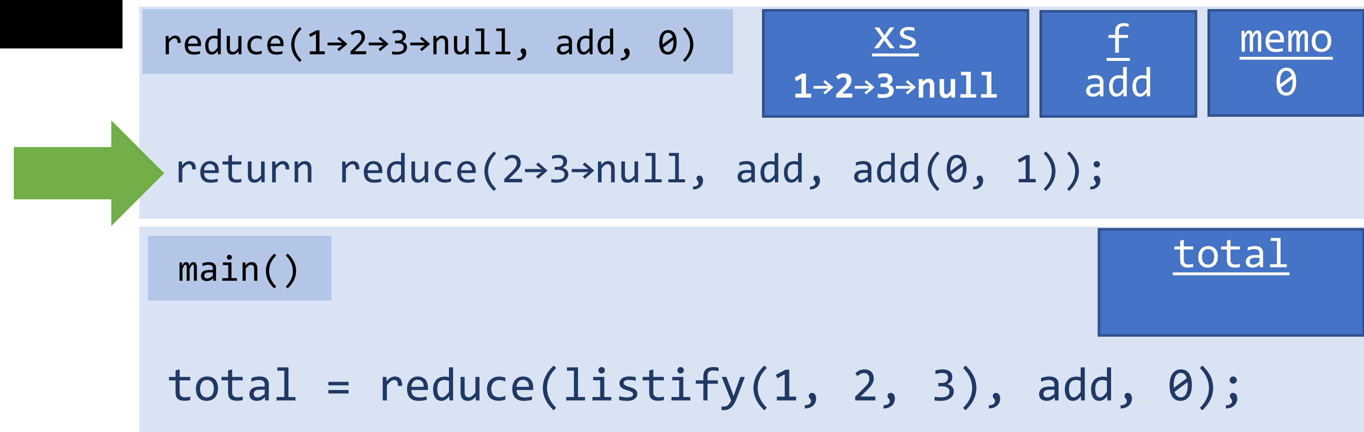
```
total = reduce(listify(1, 2, 3), add, 0);
```

Tracing **reduce**

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

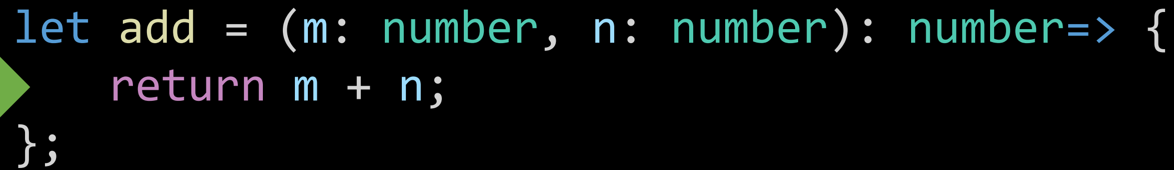
```
let add = (m: number, n: number): number=> {
    return m + n;
};
```

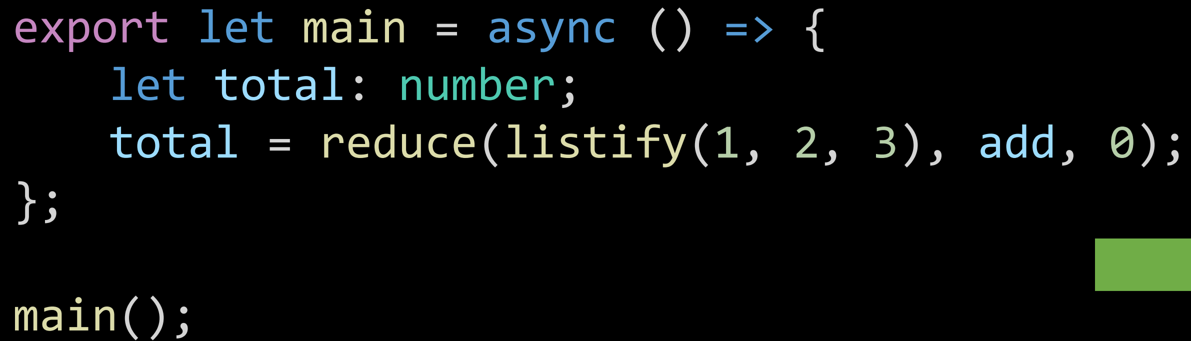
```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```



Tracing **reduce**

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





add(0, 1)

m: 0 | n: 1

return m + n;

reduce(1→2→3→null, add, 0)

xs: 1→2→3→null | f: add | memo: 0

return reduce(2→3→null, add, add(0, 1));

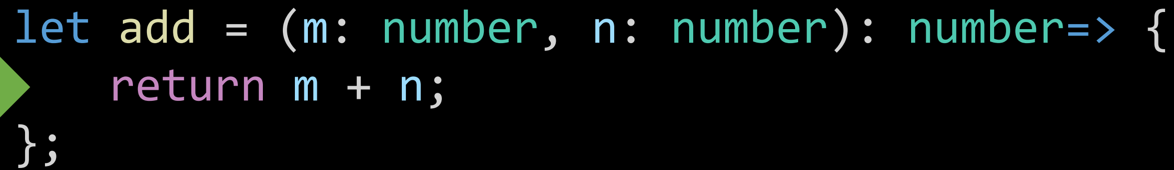
main()

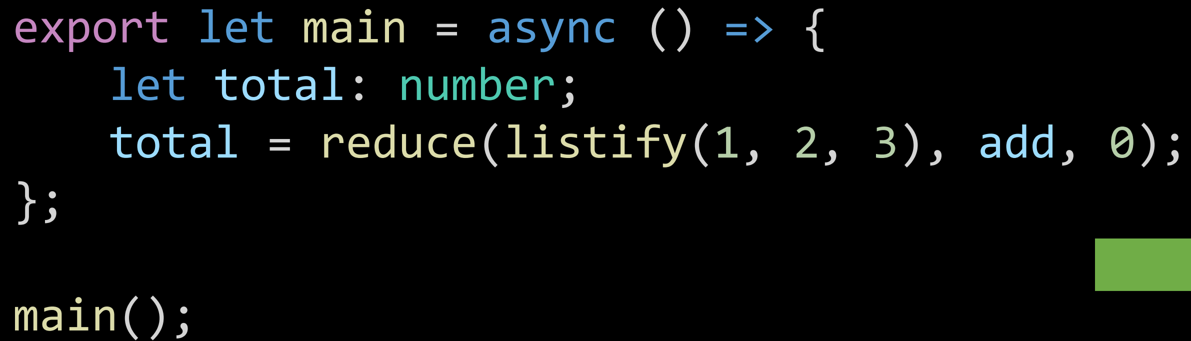
total:

total = reduce(listify(1, 2, 3), add, 0);

Tracing **reduce**

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





add(0, 1)

m	n
0	1

`return 0 + 1;`

reduce(1→2→3→null, add, 0)

xs	f	memo
1→2→3→null	add	0

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

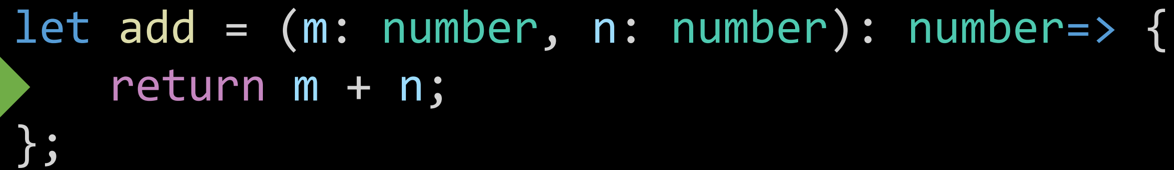
main()

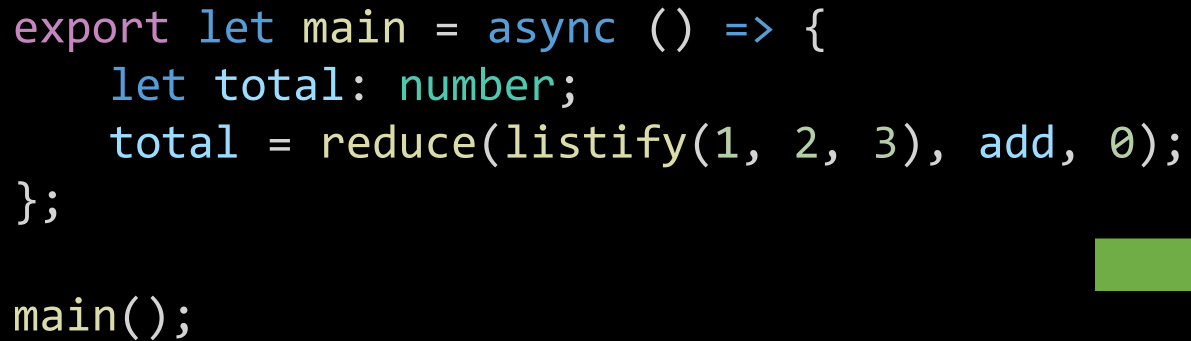
total

`total = reduce(listify(1, 2, 3), add, 0);`

Tracing **reduce**

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





add(0, 1)

m	n
0	1

return 1;

reduce(1→2→3→null, add, 0)

xs	f	memo
1→2→3→null	add	0

return reduce(2→3→null, add, add(0, 1));

main()

total

total = reduce(listify(1, 2, 3), add, 0);

Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

reduce(1→2→3→null, add, 0)	xs	f	memo
	1→2→3→null	add	0

```
return reduce(2→3→null, add, 1);
```

main()	total

```
total = reduce(listify(1, 2, 3), add, 0);
```

Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

reduce(2→3→null, add, 1)	xs 2→3→null	f add	memo 1
if (xs === null)			

reduce(1→2→3→null, add, 0)	xs 1→2→3→null	f add	memo 0
return reduce(2→3→null, add, 1);			

main()	total
total = reduce(listify(1, 2, 3), add, 0);	

Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

reduce(2→3→null, add, 1)

xs	f	memo
2→3→null	add	1

if (false)

reduce(1→2→3→null, add, 0)

xs	f	memo
1→2→3→null	add	0

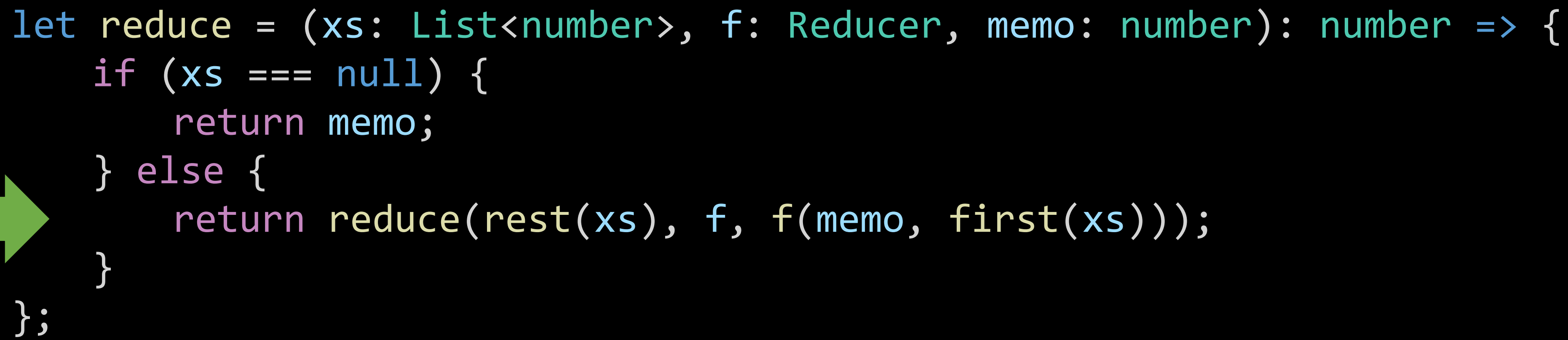
return reduce(2→3→null, add, 1);

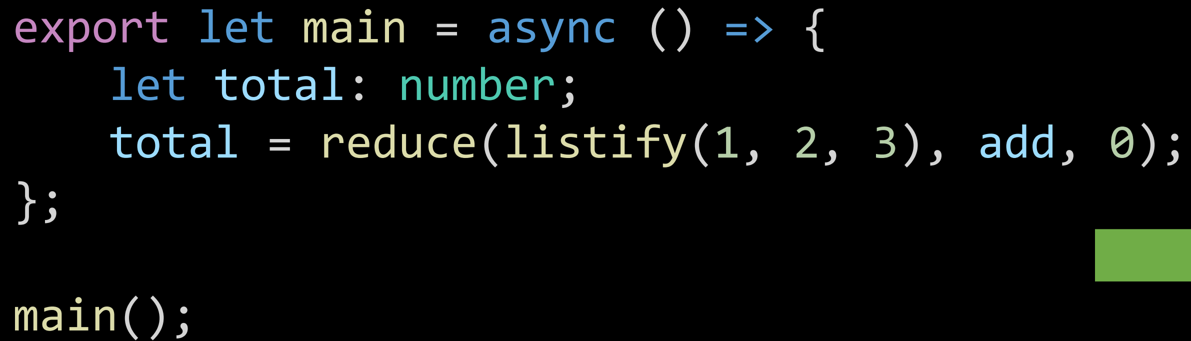
main()

total

total = reduce(listify(1, 2, 3), add, 0);

Tracing **reduce**






```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

reduce(2→3→null, add, 1)	xs 2→3→null	f add	memo 1

`return reduce(rest(xs), f, f(memo, first(xs)));`

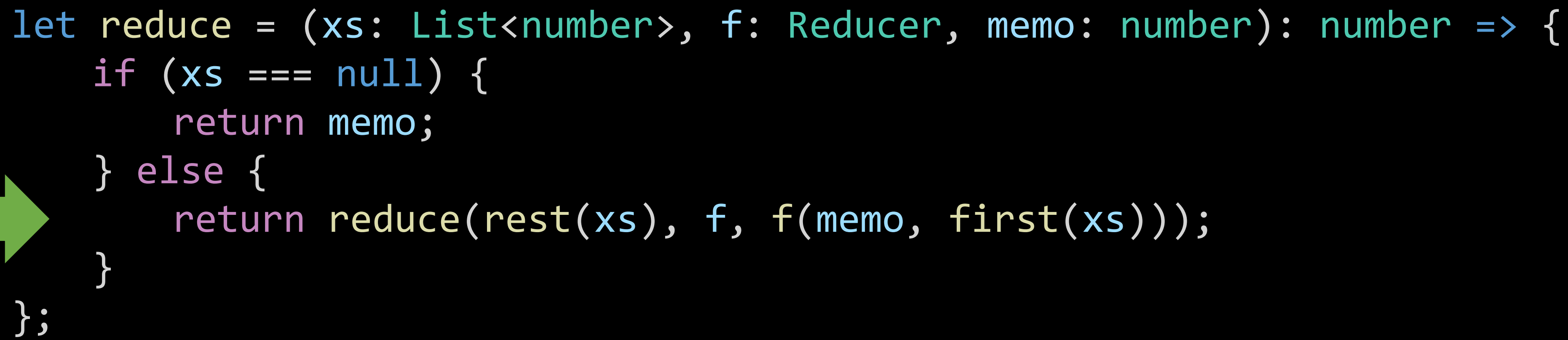
reduce(1→2→3→null, add, 0)	xs 1→2→3→null	f add	memo 0

`return reduce(2→3→null, add, 1);`

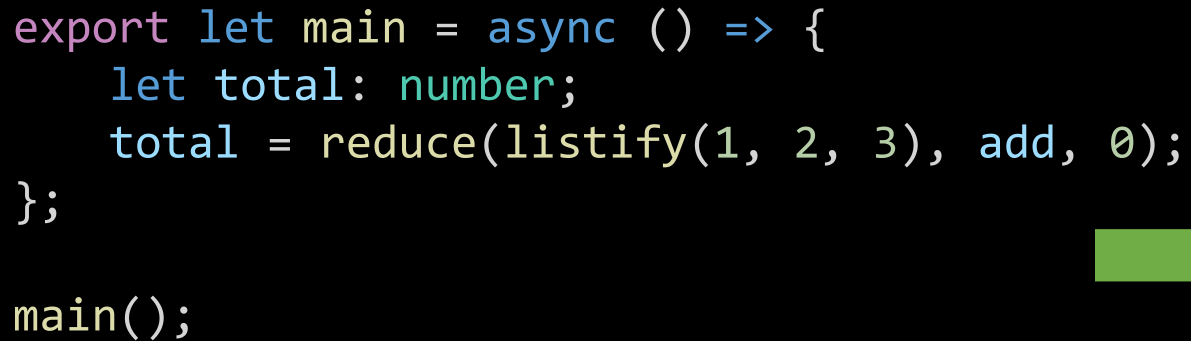
main()	total

`total = reduce(listify(1, 2, 3), add, 0);`

Tracing **reduce**




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



reduce(2→3→null, add, 1)

xs	f	memo
2→3→null	add	1

return reduce(3→null, add, add(1, 2));

reduce(1→2→3→null, add, 0)

xs	f	memo
1→2→3→null	add	0

return reduce(2→3→null, add, 1);

main()

total

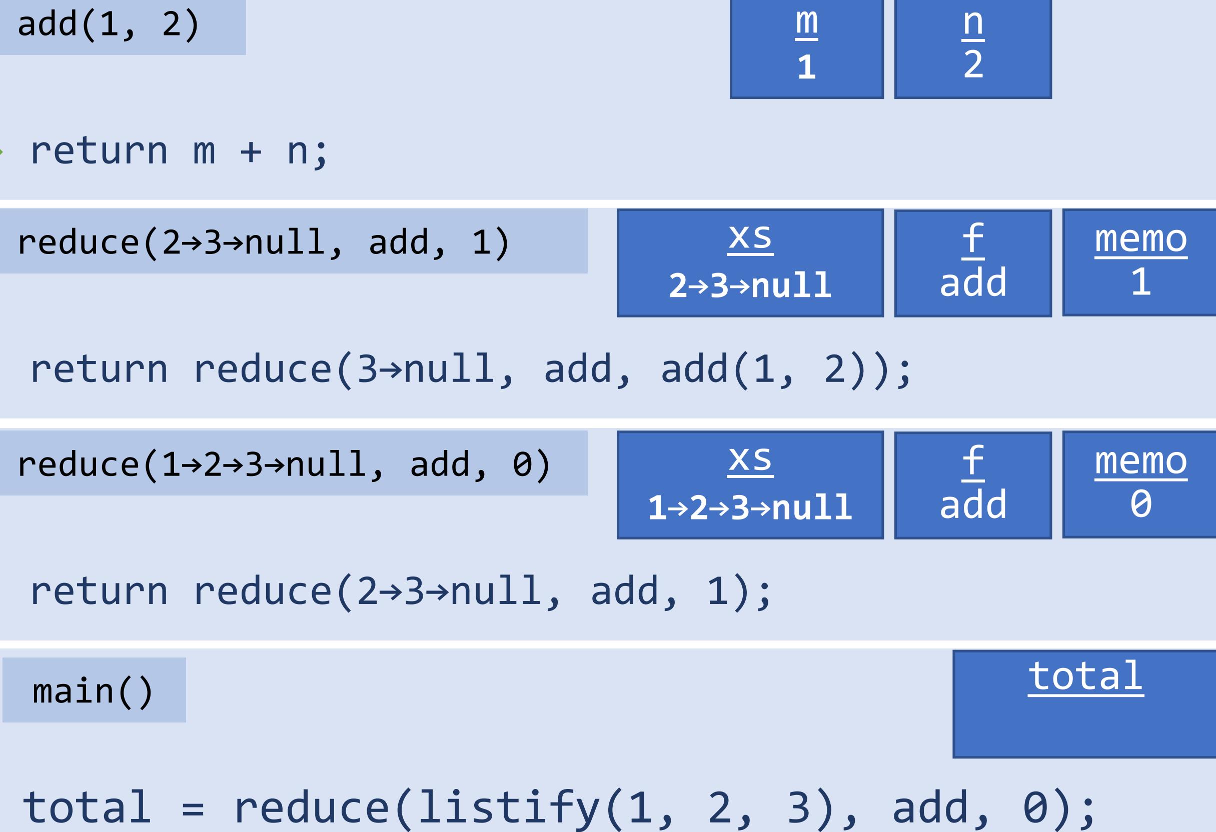
total = reduce(listify(1, 2, 3), add, 0);

Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```



Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

add(1, 2) — m: 1, n: 2

```
return 1 + 2;
```

reduce(2→3→null, add, 1) — xs: 2→3→null, f: add, memo: 1

```
return reduce(3→null, add, add(1, 2));
```

reduce(1→2→3→null, add, 0) — xs: 1→2→3→null, f: add, memo: 0

```
return reduce(2→3→null, add, 1);
```

main() — total:

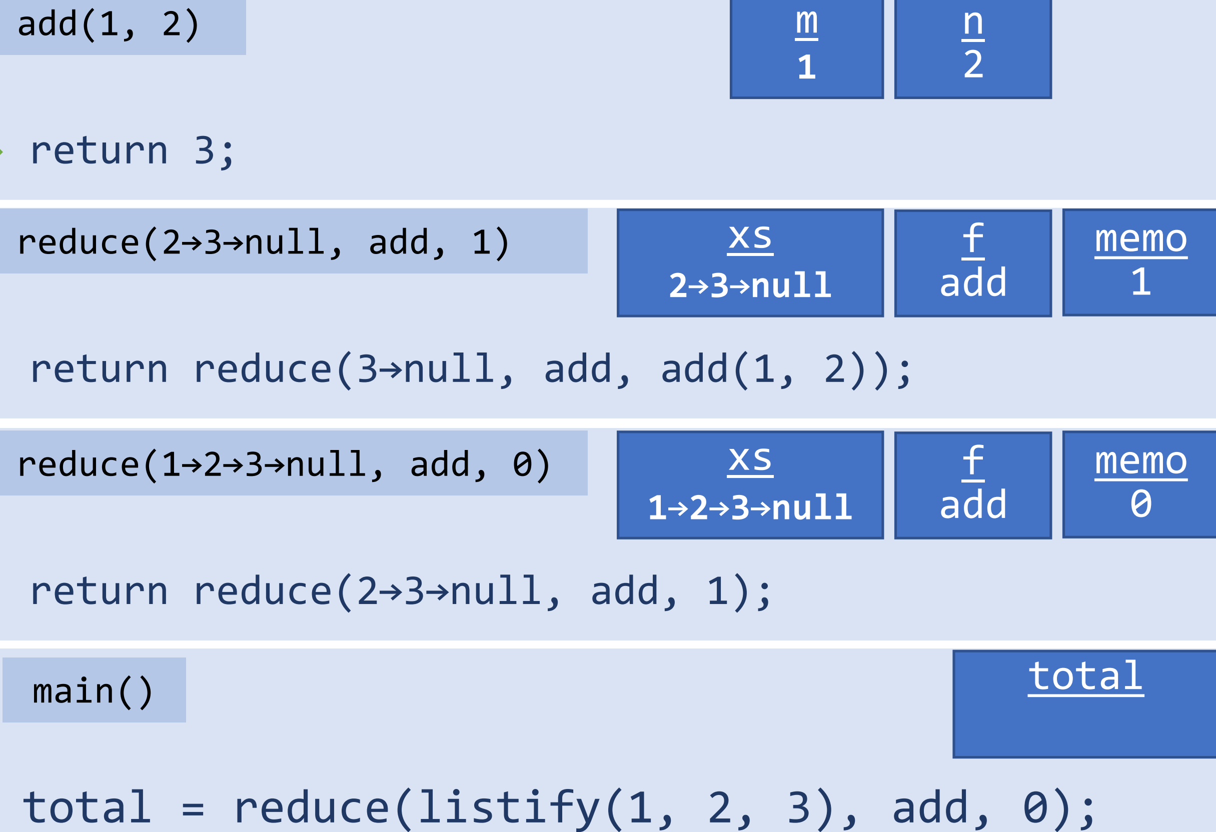
```
total = reduce(listify(1, 2, 3), add, 0);
```

Tracing **reduce**

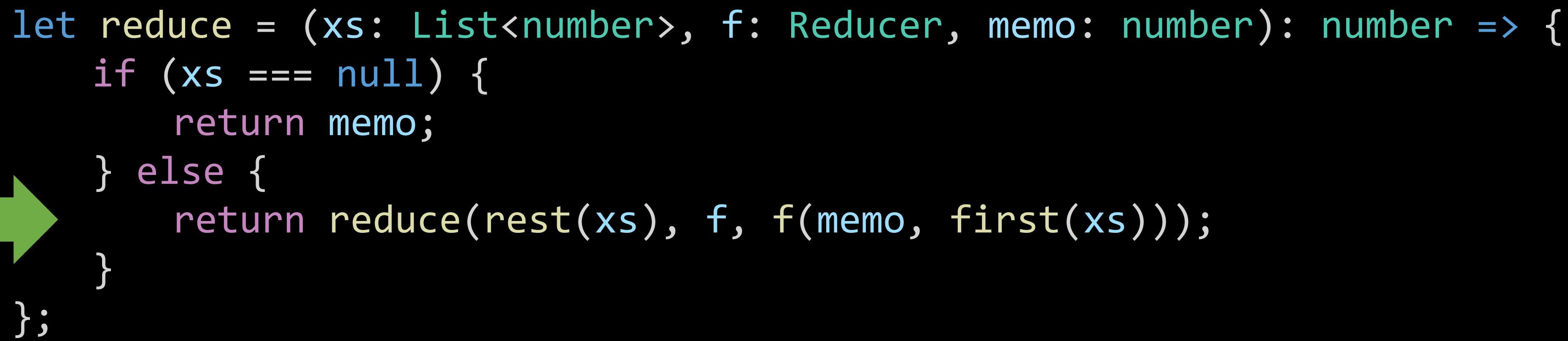
```
let reduce = (xs: List<number>, f: Reducer, memo: number): number => {
    if (xs === null) {
        return memo;
    } else {
        return reduce(rest(xs), f, f(memo, first(xs)));
    }
};
```

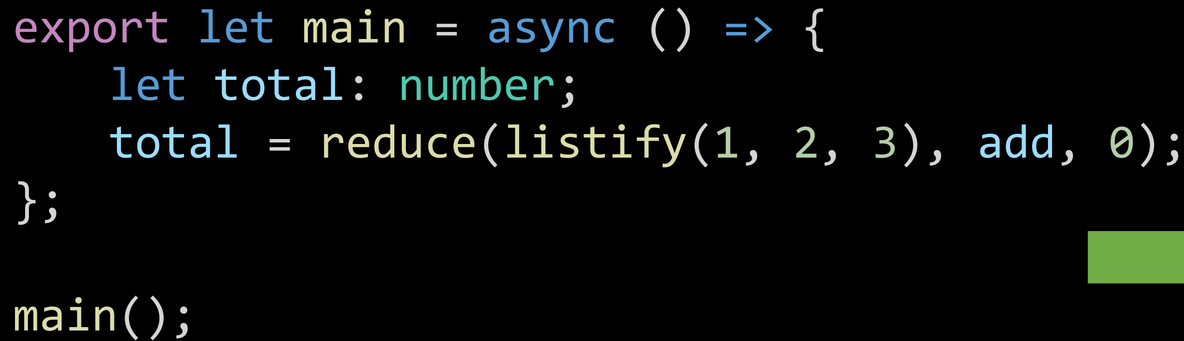
```
let add = (m: number, n: number): number=> {
    return m + n;
};
```

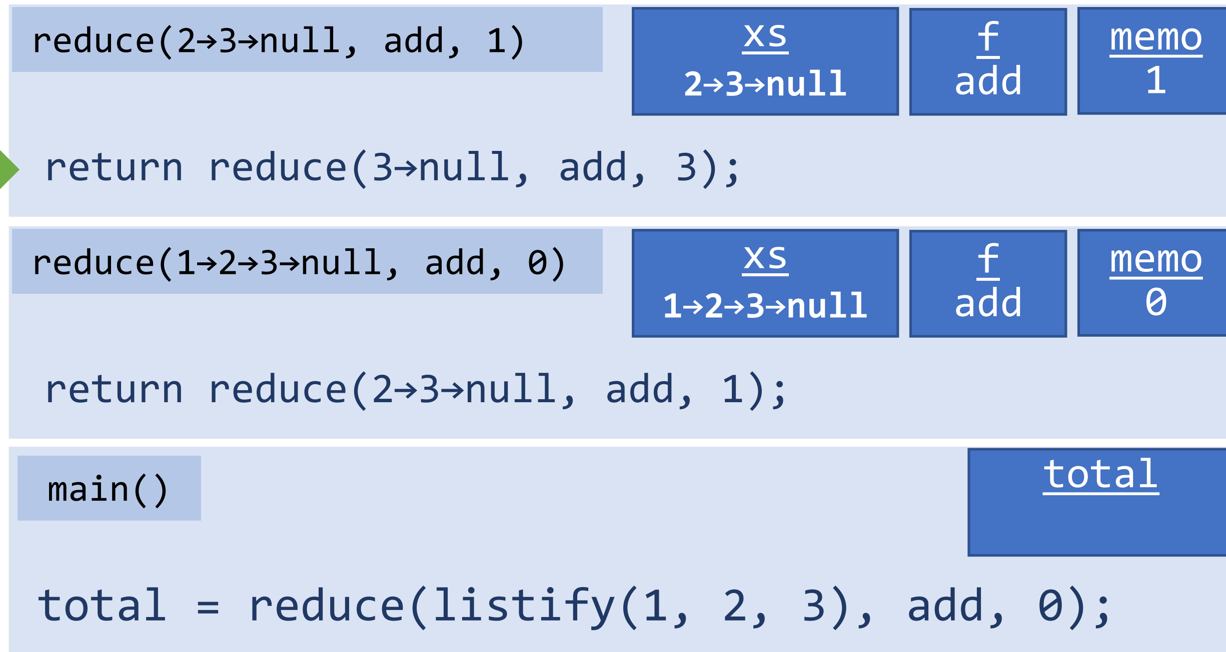
```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```



Tracing **reduce**





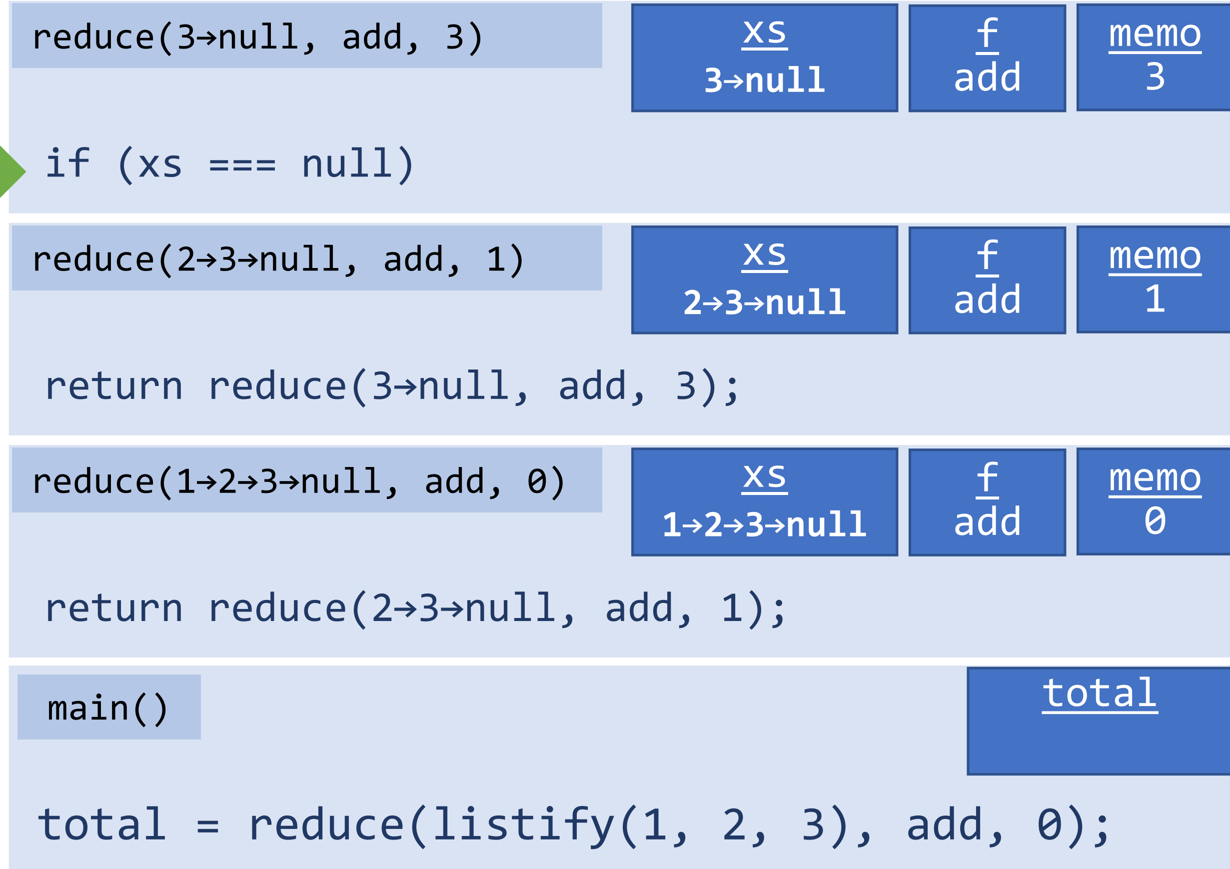


Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

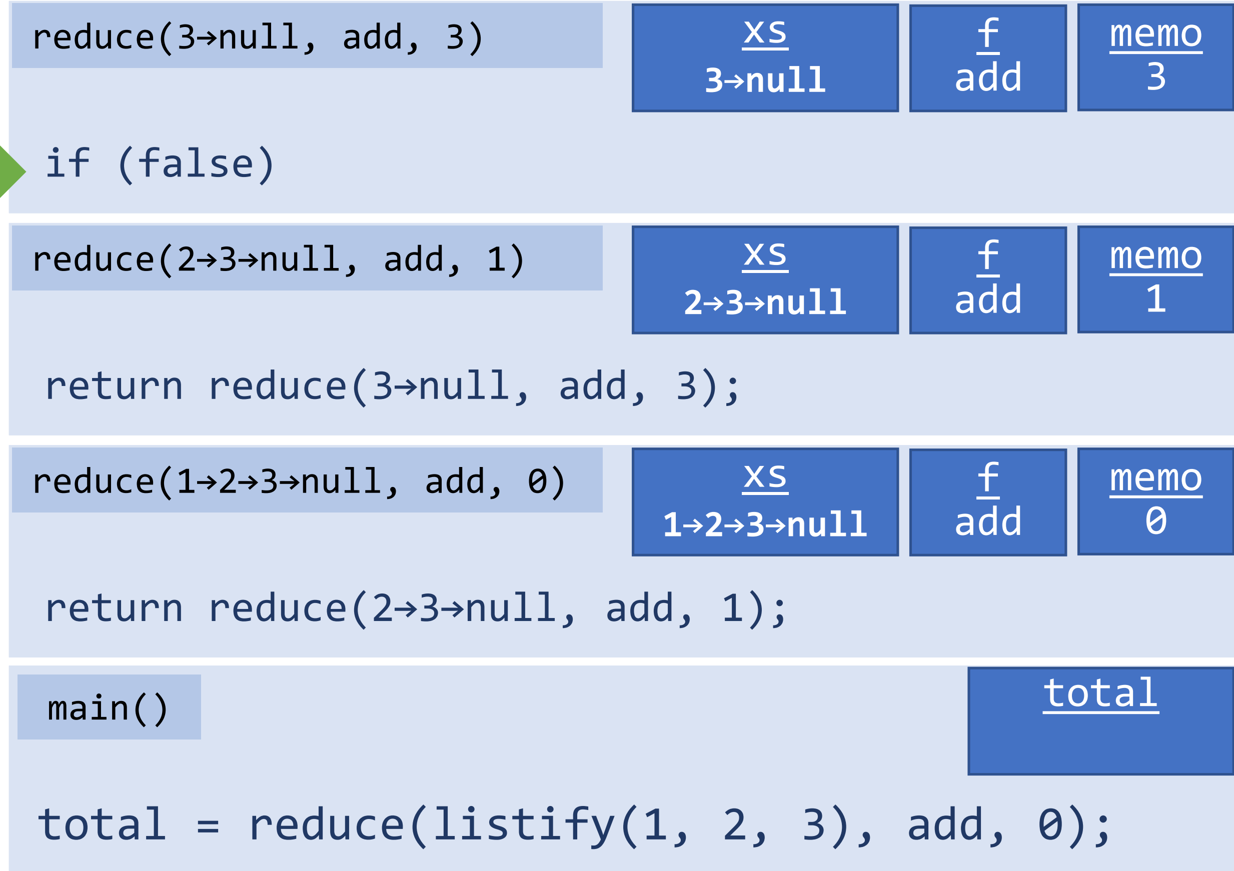


Tracing **reduce**

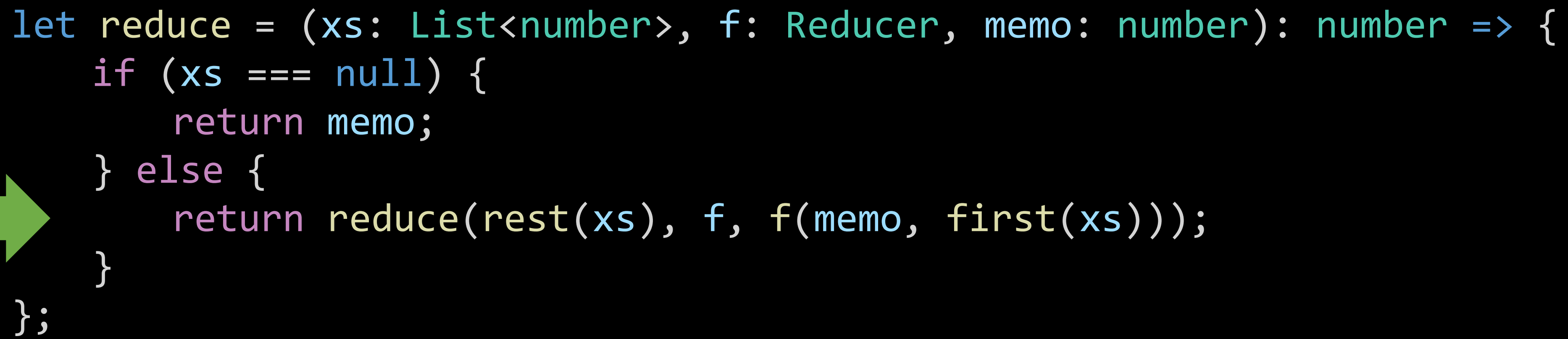
```
let reduce = (xs: List<number>, f: Reducer, memo: number): number => {
    if (xs === null) {
        return memo;
    } else {
        return reduce(rest(xs), f, f(memo, first(xs)));
    }
};
```

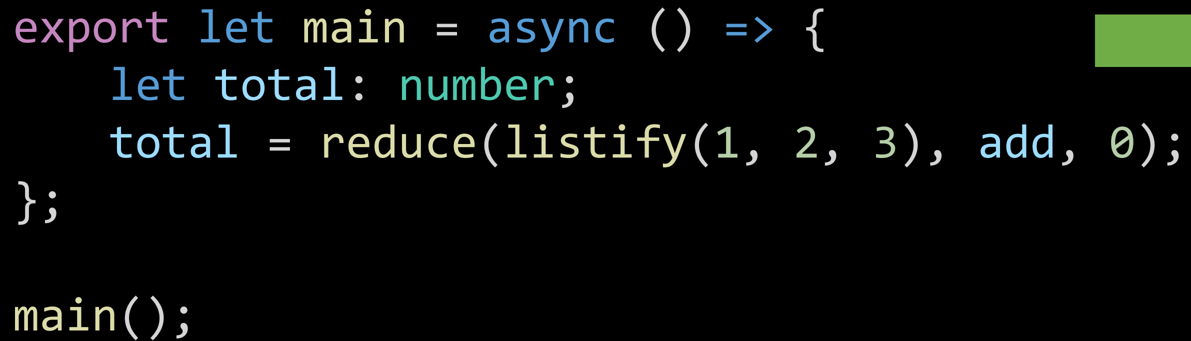
```
let add = (m: number, n: number): number=> {
    return m + n;
};
```

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```



Tracing **reduce**





reduce(3→null, add, 3)

xs	f	memo
3→null	add	3

```
return reduce(rest(xs), f, f(memo, first(xs)));
```

reduce(2→3→null, add, 1)

xs	f	memo
2→3→null	add	1

```
return reduce(3→null, add, 3);
```

reduce(1→2→3→null, add, 0)

xs	f	memo
1→2→3→null	add	0

```
return reduce(2→3→null, add, 1);
```

main()

total

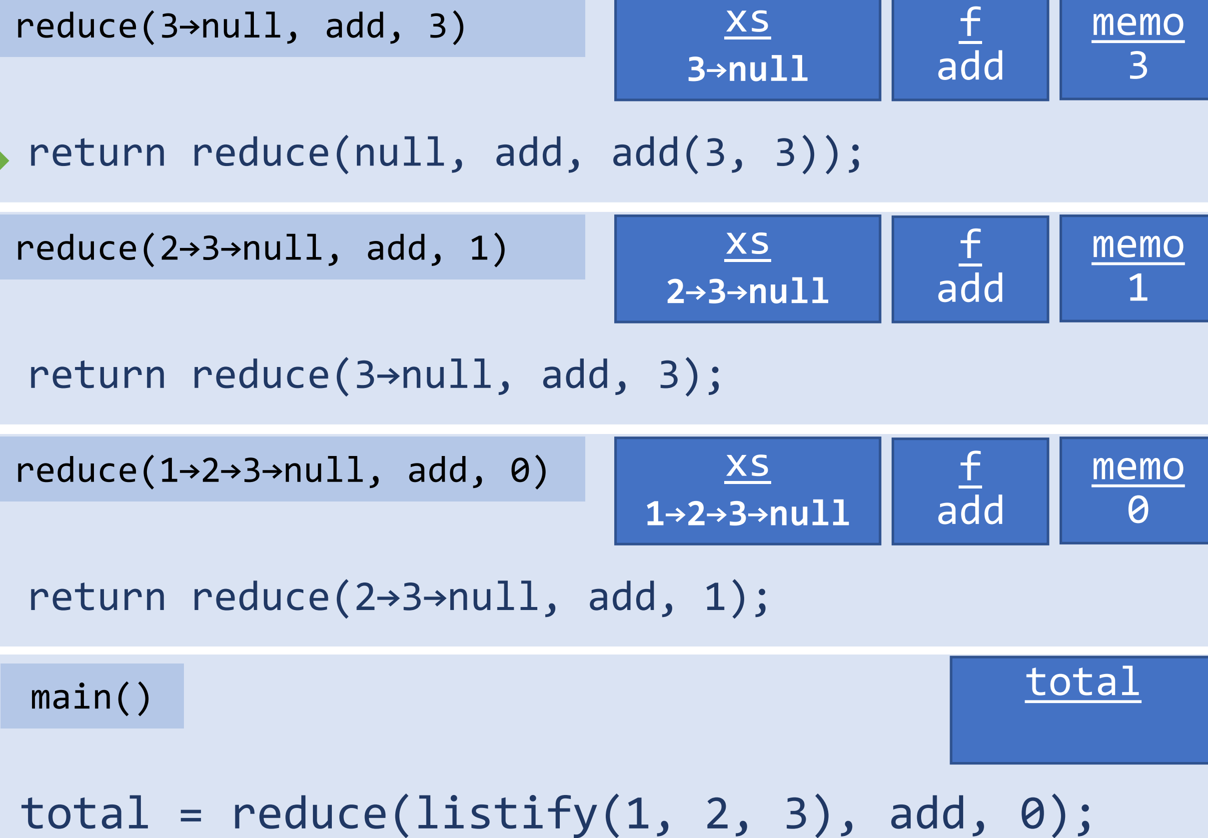
```
total = reduce(listify(1, 2, 3), add, 0);
```

Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```



Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

add(3, 3)

m	n
3	3

```
return m + n;
```

reduce(3→null, add, 3)

xs	f	memo
3→null	add	3

```
return reduce(null, add, add(3, 3));
```

reduce(2→3→null, add, 1)

xs	f	memo
2→3→null	add	1

```
return reduce(3→null, add, 3);
```

reduce(1→2→3→null, add, 0)

xs	f	memo
1→2→3→null	add	0

```
return reduce(2→3→null, add, 1);
```

main()

total

```
total = reduce(listify(1, 2, 3), add, 0);
```

Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

```
add(3, 3)                         m: 3   n: 3
  return 3 + 3;

reduce(3→null, add, 3)            xs: 3→null   f: add   memo: 3
  return reduce(null, add, add(3, 3));

reduce(2→3→null, add, 1)          xs: 2→3→null   f: add   memo: 1
  return reduce(3→null, add, 3);

reduce(1→2→3→null, add, 0)        xs: 1→2→3→null   f: add   memo: 0
  return reduce(2→3→null, add, 1);

main()                            total:
  total = reduce(listify(1, 2, 3), add, 0);
```

Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

add(3, 3)

m	n
3	3

return 6;

reduce(3→null, add, 3)

xs	f	memo
3→null	add	3

return reduce(null, add, add(3, 3));

reduce(2→3→null, add, 1)

xs	f	memo
2→3→null	add	1

return reduce(3→null, add, 3);

reduce(1→2→3→null, add, 0)

xs	f	memo
1→2→3→null	add	0

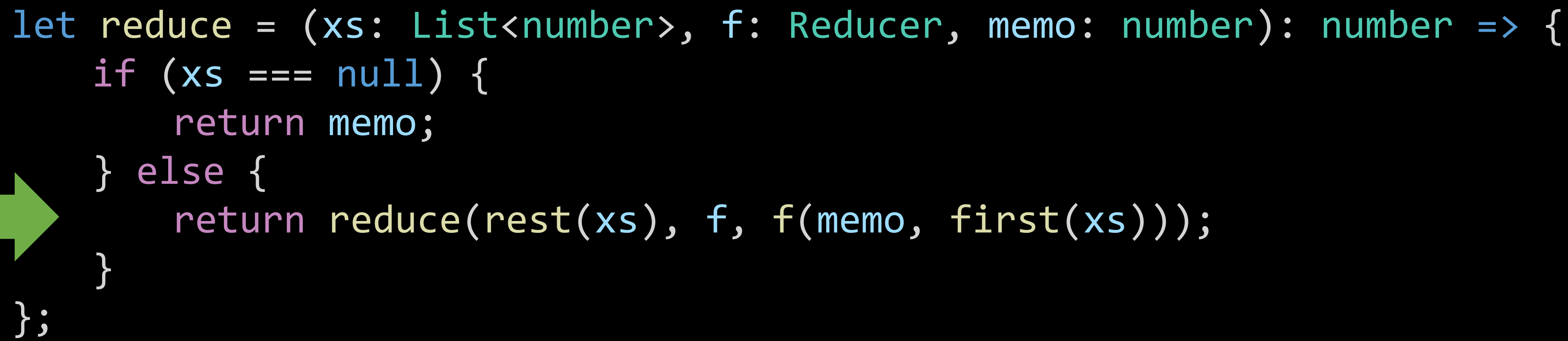
return reduce(2→3→null, add, 1);

main()

total

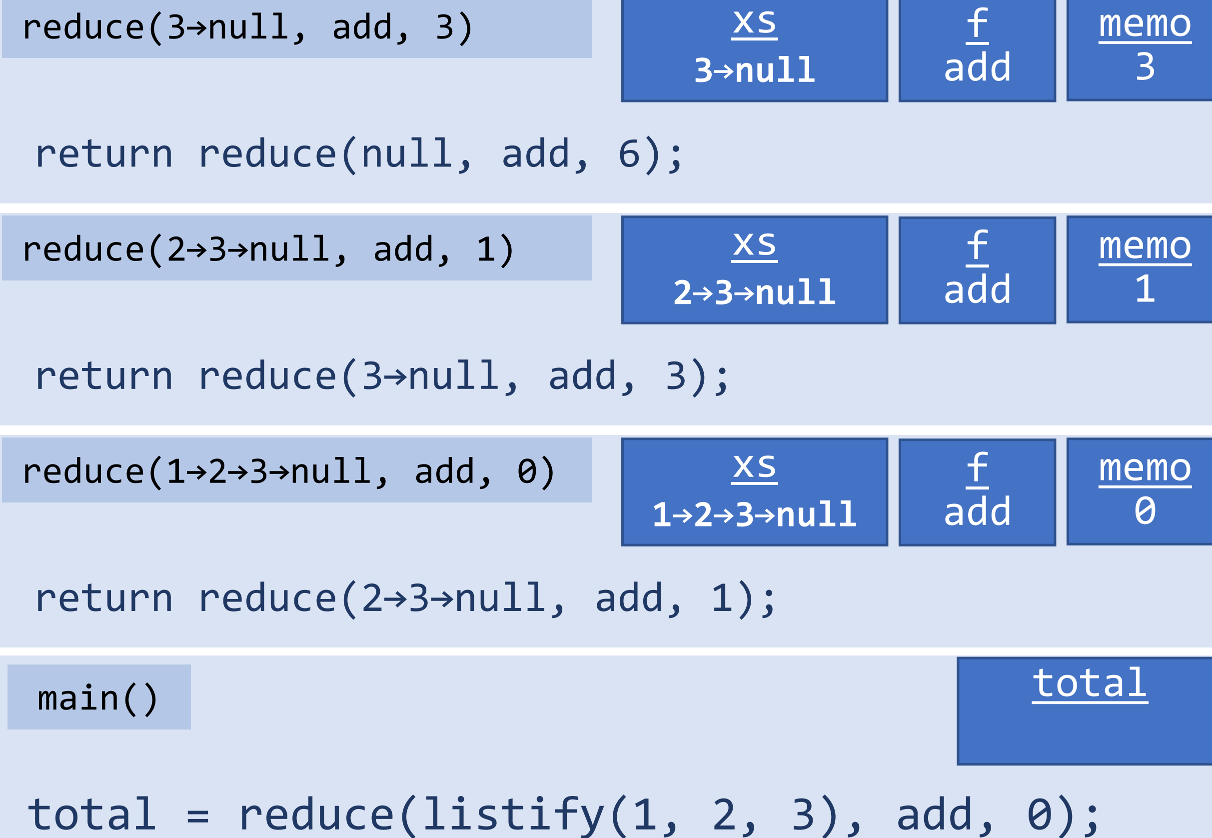
total = reduce(listify(1, 2, 3), add, 0);

Tracing **reduce**




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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```



Tracing reduce

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

Frame	xs	f	memo
reduce(null, add, 6) — if (xs === null)	null	add	6
reduce(3→null, add, 3) — return reduce(null, add, 6);	3→null	add	3
reduce(2→3→null, add, 1) — return reduce(3→null, add, 3);	2→3→null	add	1
reduce(1→2→3→null, add, 0) — return reduce(2→3→null, add, 1);	1→2→3→null	add	0

Frame	total
main() — total = reduce(listify(1, 2, 3), add, 0);	

Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

reduce(null, add, 6) — xs: null, f: add, memo: 6

```
if (true)
```

reduce(3→null, add, 3) — xs: 3→null, f: add, memo: 3

```
return reduce(null, add, 6);
```

reduce(2→3→null, add, 1) — xs: 2→3→null, f: add, memo: 1

```
return reduce(3→null, add, 3);
```

reduce(1→2→3→null, add, 0) — xs: 1→2→3→null, f: add, memo: 0

```
return reduce(2→3→null, add, 1);
```

main() — total:

```
total = reduce(listify(1, 2, 3), add, 0);
```

Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

Frame	xs	f	memo
reduce(null, add, 6) return memo;	null	add	6
reduce(3→null, add, 3) return reduce(null, add, 6);	3→null	add	3
reduce(2→3→null, add, 1) return reduce(3→null, add, 3);	2→3→null	add	1
reduce(1→2→3→null, add, 0) return reduce(2→3→null, add, 1);	1→2→3→null	add	0

Frame	total
main() total = reduce(listify(1, 2, 3), add, 0);	

Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

Frame	xs	f	memo
`reduce(null, add, 6)` — `return 6;`	null	add	6
`reduce(3→null, add, 3)` — `return reduce(null, add, 6);`	3→null	add	3
`reduce(2→3→null, add, 1)` — `return reduce(3→null, add, 3);`	2→3→null	add	1
`reduce(1→2→3→null, add, 0)` — `return reduce(2→3→null, add, 1);`	1→2→3→null	add	0

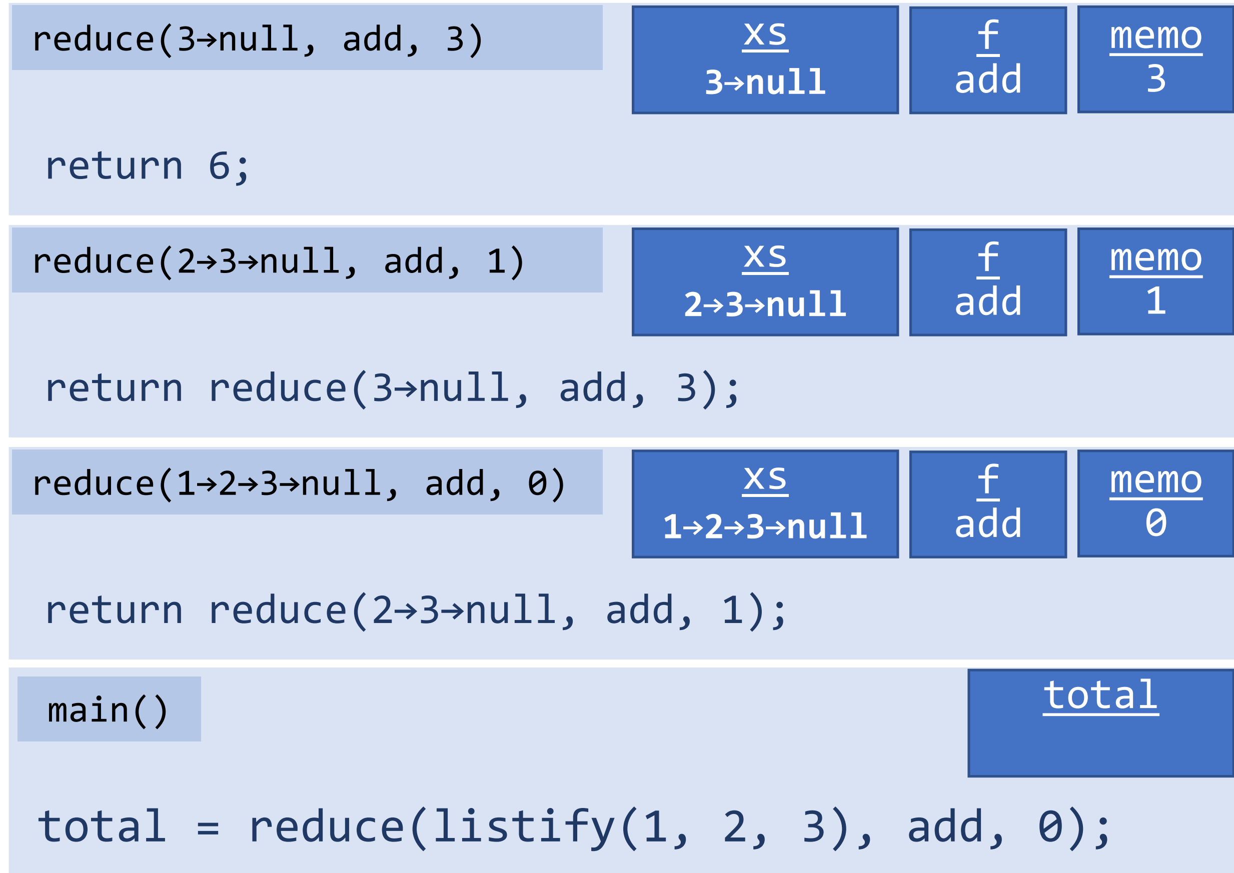
Frame	total
`main()` — `total = reduce(listify(1, 2, 3), add, 0);`	

Tracing **reduce**

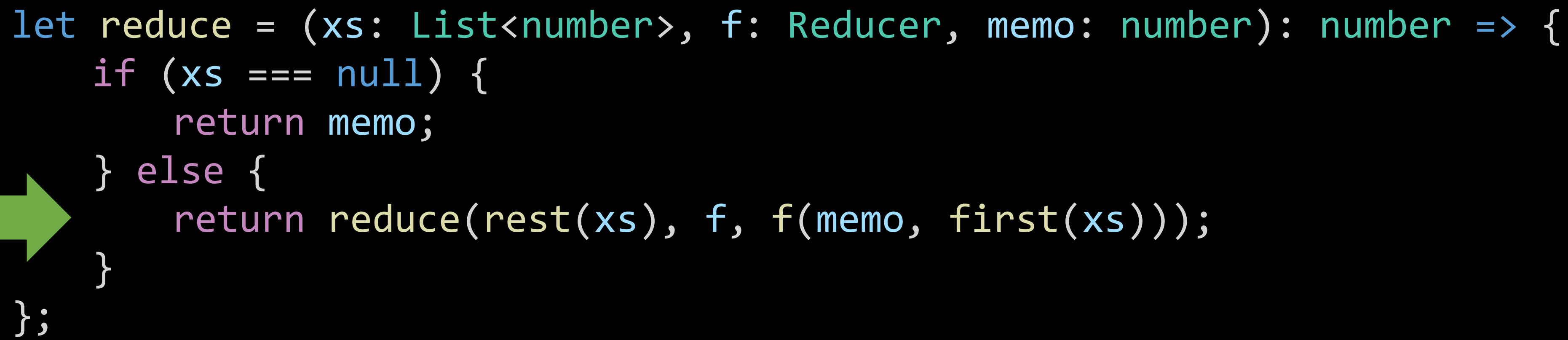
```
let reduce = (xs: List<number>, f: Reducer, memo: number): number => {
    if (xs === null) {
        return memo;
    } else {
        return reduce(rest(xs), f, f(memo, first(xs)));
    }
};
```

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```



Tracing **reduce**




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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

reduce(2→3→null, add, 1)

xs	f	memo
2→3→null	add	1

return 6;

reduce(1→2→3→null, add, 0)

xs	f	memo
1→2→3→null	add	0

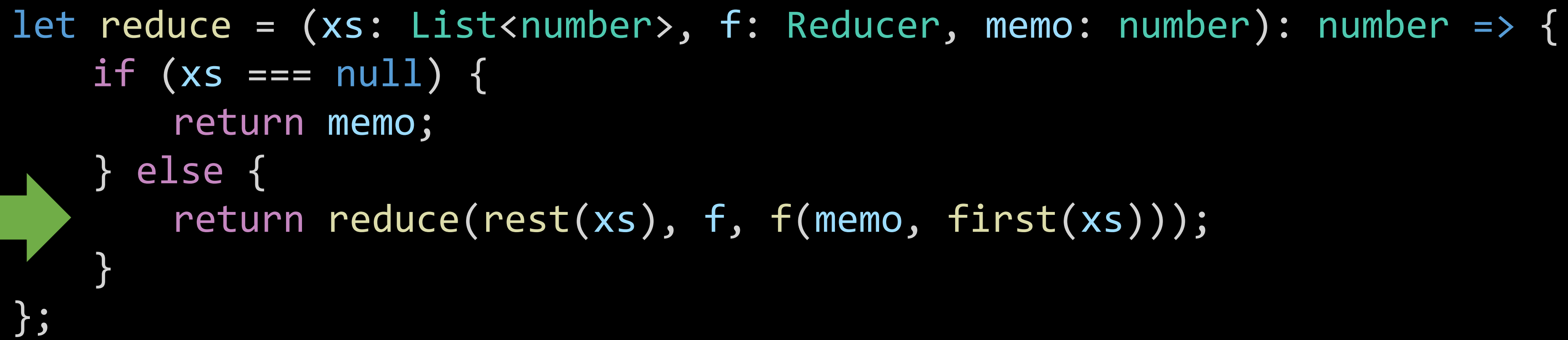
return reduce(2→3→null, add, 1);

main()

total

total = reduce(listify(1, 2, 3), add, 0);

Tracing **reduce**




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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

reduce(1→2→3→null, add, 0)

xs	f	memo
1→2→3→null	add	0

```
return 6;
```

main()

total

```
total = reduce(listify(1, 2, 3), add, 0);
```

Tracing **reduce**

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

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

```
export let main = async () => {
    let total: number;
    total = reduce(listify(1, 2, 3), add, 0);
};

main();
```

main()

total
6

```
total = 6;
```

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

- Open lec13 / 01-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 (Reducer) function named ***max*** that is giventwo pare 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)** In the main function, **assign** to the variable ***points*** the result of mapping the ***gamesToPoints*** function over the ***games*** list.
    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 - 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);
```

Breaking a Project into Multiple Files

- You can **export** *functions* and *classes* from one TypeScript file

```
export let f = (…

export class {…
```

- And **import** them into *another* TypeScript file

```
import { <names>, <of>, <functions/classes> } from "./<file>";
```

- Example:

```
import { filter, map, reduce } from "./list-utils";
```

Referencing Imported Files

- Where you see the dot-slash in **"./library"**, this means **"from within the same folder I am in, import the library.ts file"**
  - Note: you do not specify the .ts file extension

- To move up a folder, the dot-dot-slash in **"../super"**, means, **"move up to my parent folder, and import *super.ts* from there"**

- To move up to a parent folder, and back down to a sibling folder: **"../ps01-adventure/index-app"** to reference the file index-app.ts in Problem Set 01.

- No stress: when you need to do imports/exports in problem sets, we'll guide you through it.

Library Files

- As we move further into the semester we will break our projects into library files whose classes and functions can be reused by many different apps.

- Let's take a look at an example library file in lec13 / list-utils.ts

Hands-on: filter/map/reduce Pipeline

- Open **`02-stats-app.ts`**

1. First, `import` the functions `filter`, `map`, and `reduce` from ./list-utils.ts:

```
import { filter, map, reduce } from "./list-utils";
```

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

```
let filtered: List<Game> = filter(games, PREDICATE);
```

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

```
let values: List<number> = map(filtered, TRANSFORM);
```

4. 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);
```

5. 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.

```
import { filter, map, reduce } from "./list-utils";
```

```
// TODO #1
let filtered: List<Game> = filter(games, fewPoints);
// TODO #2
let values: List<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 ____ , what was the ____ ____

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

- points
- assists
- fouls
- blocks
- etc

- total
- average
- min
- max
- 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: (# Filters) x (# Maps) x (# Reduces) different analyses.

Type Inference: Variables

- Can you fill in the blank correctly so that there is not an error?

```
let x: _________ = "hello, world";
```

- It's a *string!* As a human, we can *infer* this based on its context in a variable initialization statement.

- Why can't the programming language infer this?

- It turns out… *it can!*

Type Inference: Variables

- TypeScript, and most modern programming languages, can infer the type of a variable based on how it is initialized.

- When you write… **`let x = 0;`**
  … TypeScript infers that you intended to write **`let x: number = 0;`**

- If you later tried to assign **`x`** a value that is not a number, you will get a type error.

- To use type inference with variables in COMP110:
  You must initialize your variable in the same statement you declare it.

- If you do not declare and initialize at the same time, you should specify the type like you've been doing up until now.

Follow-along

- Open 03-type-inference-app.ts
- Under the **TODO**, remove the explicit type declarations from the variable declarations

```
let games = await csvToList("Game Data", Game);
let filtered = filter(games, isWin);
let values = map(filtered, toPoints);
let result = reduce(values, sum, 0);
```

- Notice there are no errors!
- If you hover your cursor over any variable you will be told exactly what type TypeScript has inferred your variable to be.

Plot Twist: We've relied on variable Type Inference all semester long...

- Whenever we've declared a function, that function is assigned to a variable whose type TypeScript has inferred for us.

- Just like we previously explicitly typed data variables, we can explicitly type function variables, as well:

```
let isWin: Predicate<Game> = (g: Game): boolean => { …
```

```
let toPoints: Transform<Game, number> = (g: Game): number => { …
```

```
let sum: Reducer<number, number> = (m: number, n: number): number => { …
```

- This even more *redundant and verbose* than variable declarations. Isn't type inference awesome?