Recursive Data Types, Linked Lists and Abstraction Barriers

What is returned by a call to **`odd(6)`**?

```
let odd = (n: number): number => {
   if (n % 2 === 1 && n % 3 === 0) {
      return n;
   } else {
      return odd(n + 1);
   }
};
```

Announcements

- Worksheet goes out tonight and due Sunday 10/27 at 11:59pm!
- Next problem set will go out by Thursday and be focused on recursion and linked lists (today's topic!)
- Next graded warm-up questions are Thursday and will be on Lecture 12 and 13 (recursion and recursive data types)
- Quiz is Tuesday 10/30

Compound Data Type Properties

- So far we've focused on classes with value-type properties, such as:
  - string
  - number
  - boolean
- Properties can also be reference types, like:
  - arrays
  - objects

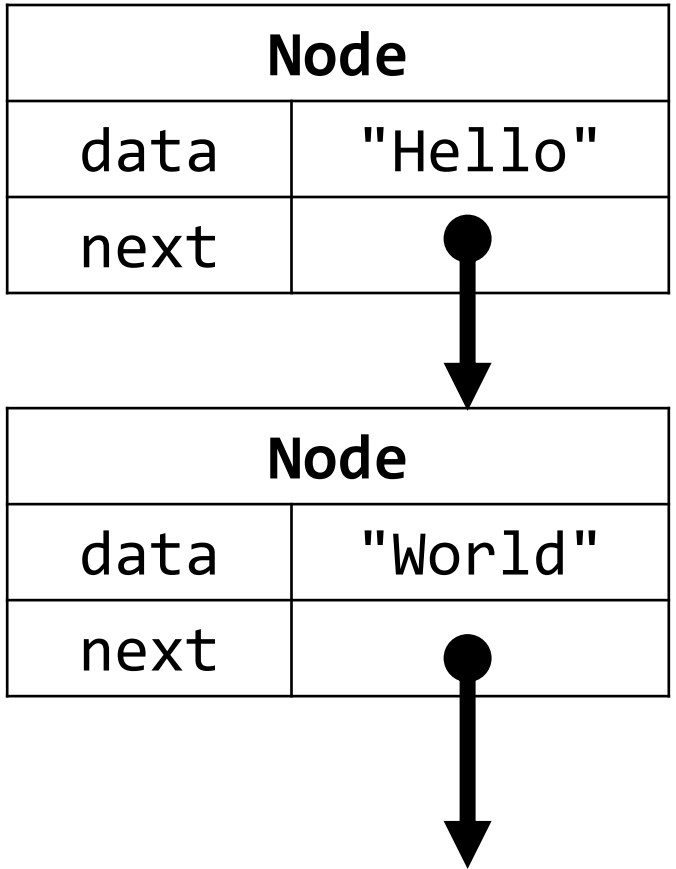
```
class Person {
    name: string = "";
    pets: Dog[] = [];
}

class Dog {
    name: string = "";
    breed: string = "";
}
```

Recursive Data Types

- A property *can* refer to another object of the *same type*
- Notice the class **Node**. It has a property named **next** and its value must be... *another Node*.
- This is a recursive data type!
- We'll discuss how to initialize a recursive property to avoid infinite recursion shortly...

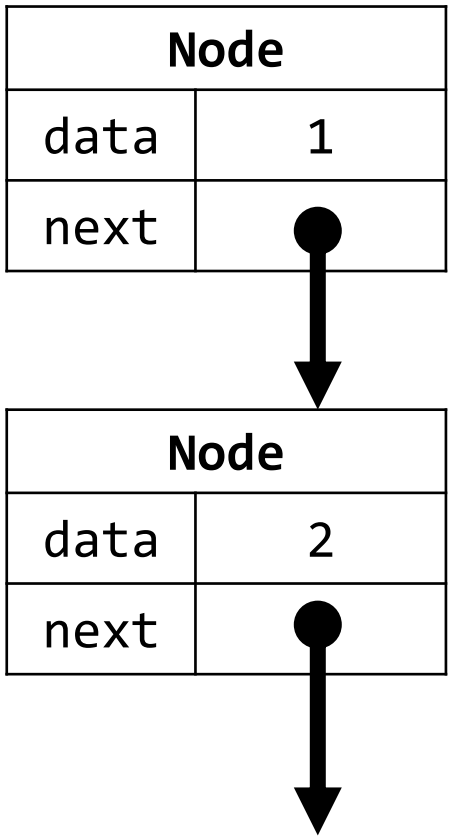
```
class Node {
   data: string = "";
   next: Node = ?;
}
```



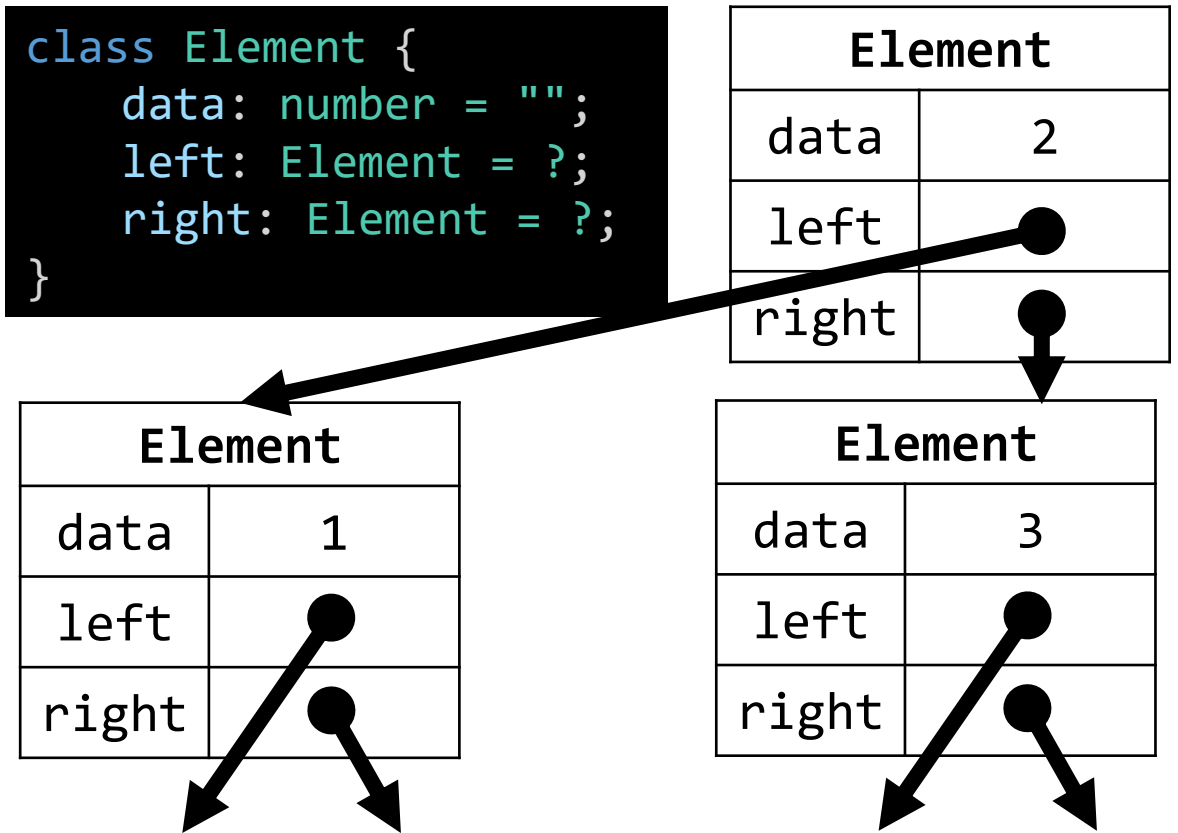
Data Structures

- You can use this ability to form **data structures** with different properties and uses.
- In COMP110, you'll explore the Linked List (left)
- In COMP410, you'll explore other data structures like Trees (right) and Graphs

```
class Node {
    data: number = "";
    next: Node = ?;
}
```




```
class Element {
    data: number = "";
    left: Element = ?;
    right: Element = ?;
}
```

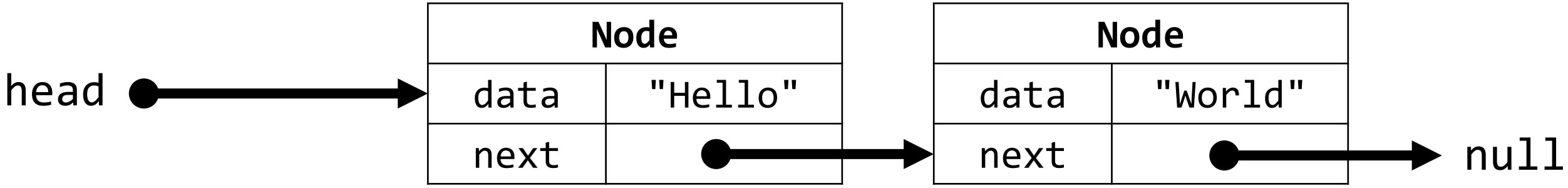


Linked List

- The classic, simple data structure in Computer Science

- Formed by chaining together a sequence of objects
  - The first node is conventionally called the **head**

- Linked lists are more cumbersome to work with than arrays
  - They're vitally important for understanding and exploring fundamentals including:
  - **`null`** values
  - References
  - Recursive algorithms

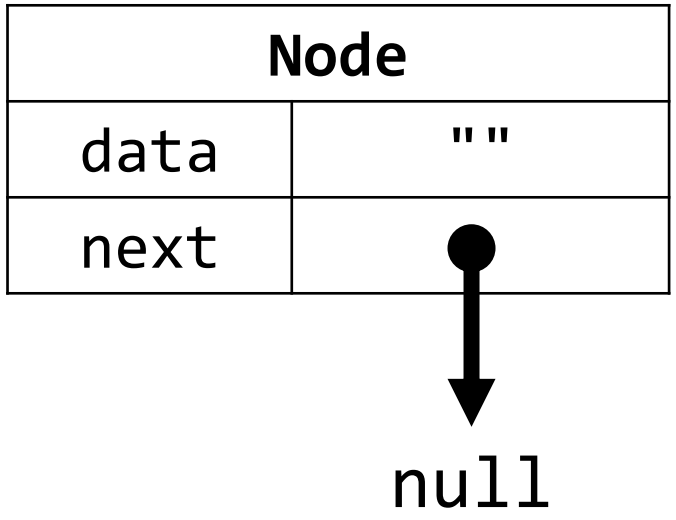


What is a recursive property's "base case"?

- If a `Node` refers to a `next` `Node`, and the `next` `Node` refers to another `next` `Node`, then *when does it end?*

```
class Node {
   data: string = "";
   next: Node = null;
}
```

- Recursive properties are terminated with a special value called **`null`**.
  - It is a "reference to nowhere" that you can read as "this property refers to nothing."



- Our linked lists are "**null** terminated".

Hands-on: Constructing a Linked List Node-by-Node

- Open 00-node-app.ts

1. Before you begin, understand the two Nodes already constructed. Try diagramming these out on paper.

2. At TODO #1 - Construct a new Node object with the properties described in the comments. Assign it to `head`.

3. At TODO #2 - Print the value of the last Node in the list that should now contain the data "C".

- Check-in when you've got UNC printing out!

Costs of not abstracting away details...

- The last hands-on suffers from two important problems:

1. It's cumbersome and error prone to construct a linked list by manipulating the properties of its nodes directly.

2. Our program is now tightly coupled to the details of the Node class' properties. Changing the implementation of Node would permeate all aspects of our program that relied upon a Node.

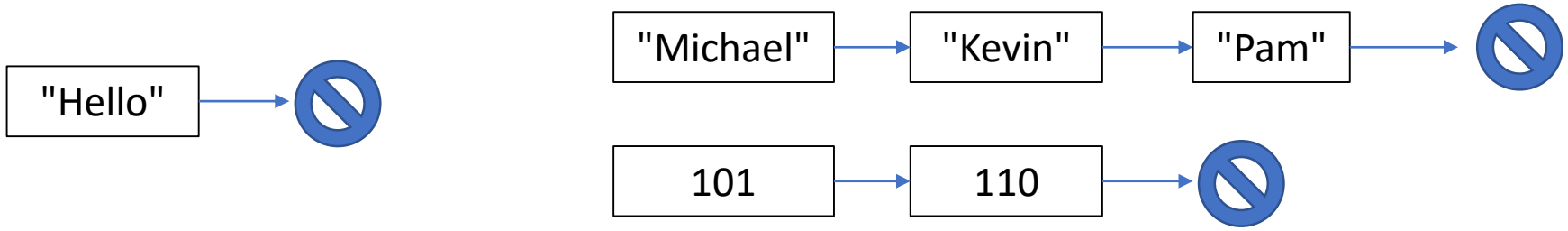
- These are the costs of programming at the wrong level of abstraction.

- Let's try working with the abstract concept of a "list" instead...

What is a Linked List?

1. A **List** may be empty

2. A List may be a sequence of one or more values of the same type



3. Each item in a List is called a **Node**

4. The end of a List is marked by a special value called **`null`**

What can you *do* with a list?

1. You can *construct* a new node at the front of another list
   - via the ***cons*** function

2. You can ask a list for its first value
   - via the ***first*** function

3. You can ask a list for a sub-list of itself, excluding the first value
   - via the ***rest*** function

- That's it! By default, these are the only operations you can do with a list!
  - These are all the capabilities you *need.*
  - Using these simple operations, you will write more advanced functions, or abstractions, to perform more sophisticated tasks with lists.

cons-tructing a list, value-by-value (1/2)

- The function **cons** is short for "construct List".

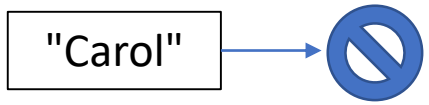
- The **cons** function requires 2 parameters:
  1. The value you are adding on to the front of the `List`
  2. The `List` you are adding the value onto

- The **cons** function *returns* a new `List` with the value added to the front.

The **cons** function **usage** (2/2)

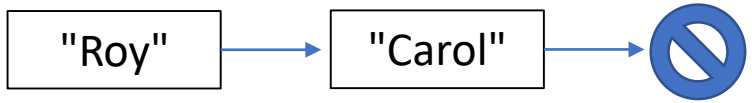
- Construct a list with a single Node in it

```
let names = cons("Carol", null);
```



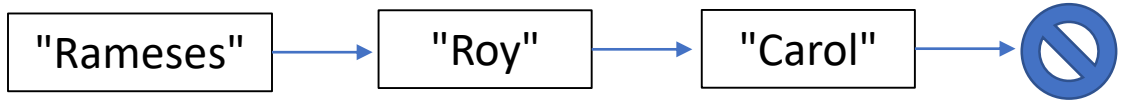
- Construct a list with two values in it

```
names = cons("Roy", cons("Carol", null));
```



- Modify a list by adding onto itself

```
names = cons("Rameses", names);
```



Follow-along: 01-list-abstraction-app.ts

```
let list = cons("N", cons("C", null));
print(toString(list));

// TODO: cons U onto list
list = cons("U", list);
print(toString(list));
```

The **`first`** function returns the first value of a list

- The List function **`first`** returns the first value in a non-empty list
  - Warning: the `first` function *will* error if given an empty List

- The **`first`** function requires one parameter: a *non-empty* list

- Usage:

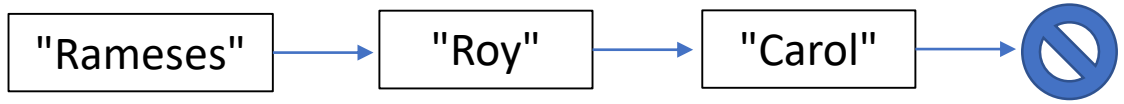
```
let series = cons(10, cons(20, cons(30, null)));
print(first(series));
```

→ 10

`first` Examples Visualized

Function Call

Return Value

`first(` 

`)`

"Rameses"

`first( null )`

ERROR

The **`rest`** function returns a sub-List, w/o first value

- The function **`rest`** returns a list with every value except the first
  - Warning: the `rest` function *will* error if given an empty `List`

- The **rest** function requires one parameter: a non-empty List.

- Usage:

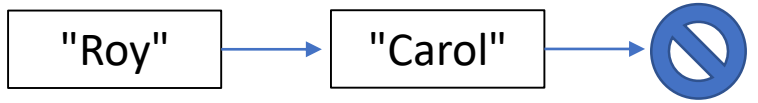
```
let series = cons(10, cons(20, cons(30, null)));
print(toString(rest(series)));
```

```
20 → 30 → null
```

`rest` Examples Visualized

Function Call

Return Value



ERROR

Follow-Along: Print the 2nd & 3rd Entries in the List

```
// TODO: Print the 2nd and 3rd Values
print(first(rest(list)));
print(first(rest(rest(list))));
```

How are `cons`, `first`, and `rest` implemented?

- They're defined in list.ts
- These are very simple functions!
- What's the big deal?
- We've addressed the two shortcomings of our initial example! How?

```
/* Constructor */
export let cons = (data: string, next: Node): Node => {
    let n = new Node();
    n.data = data;
    n.next = next;
    return n;
};

/* Selectors */
export let first = (n: Node): string => {
    return n.data;
};

export let rest = (n: Node): Node => {
    return n.next;
};
```

The **count** Algorithm: counting values of a `List`

- How can we write a function that, given a List of any length, we can count the number of elements in it?

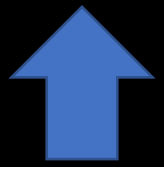
- Let's try it with ***pseudo-code*** first!

- **Count Algorithm**, Given any List
  1. *If* the List is empty, *then* the `count` is 0
  2. *Else*, `count` is 1 + the count algorithm applied to *the rest of* the List

- UTA Live Demo

count in code

```
let count = (list: Node): number => {
   if (list === null) {
      return 0;
   } else {
      return 1 + count(rest(list));
   }
};
```

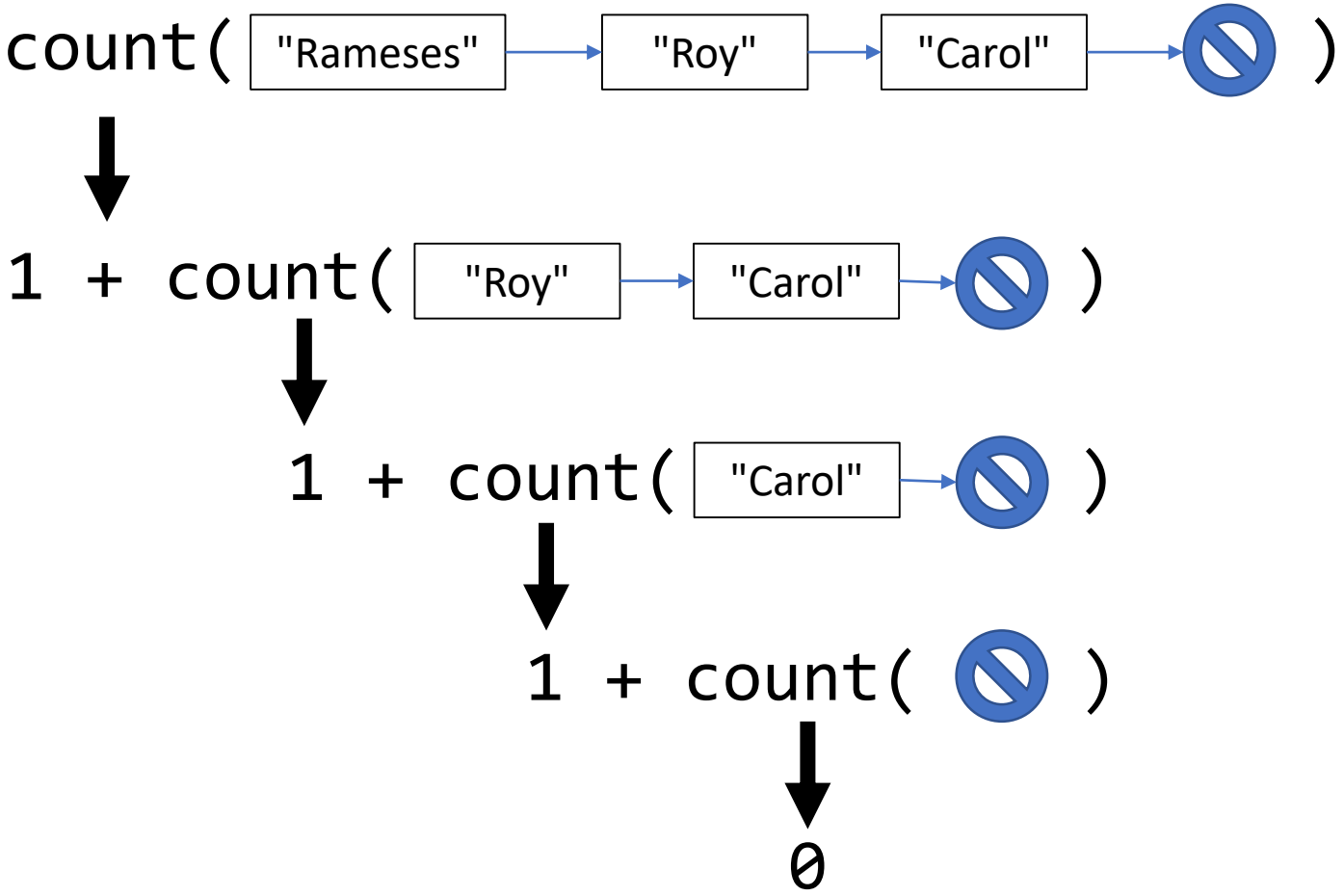


- What magic is this?

- Recursion! The count function is defined *in terms of itself.*

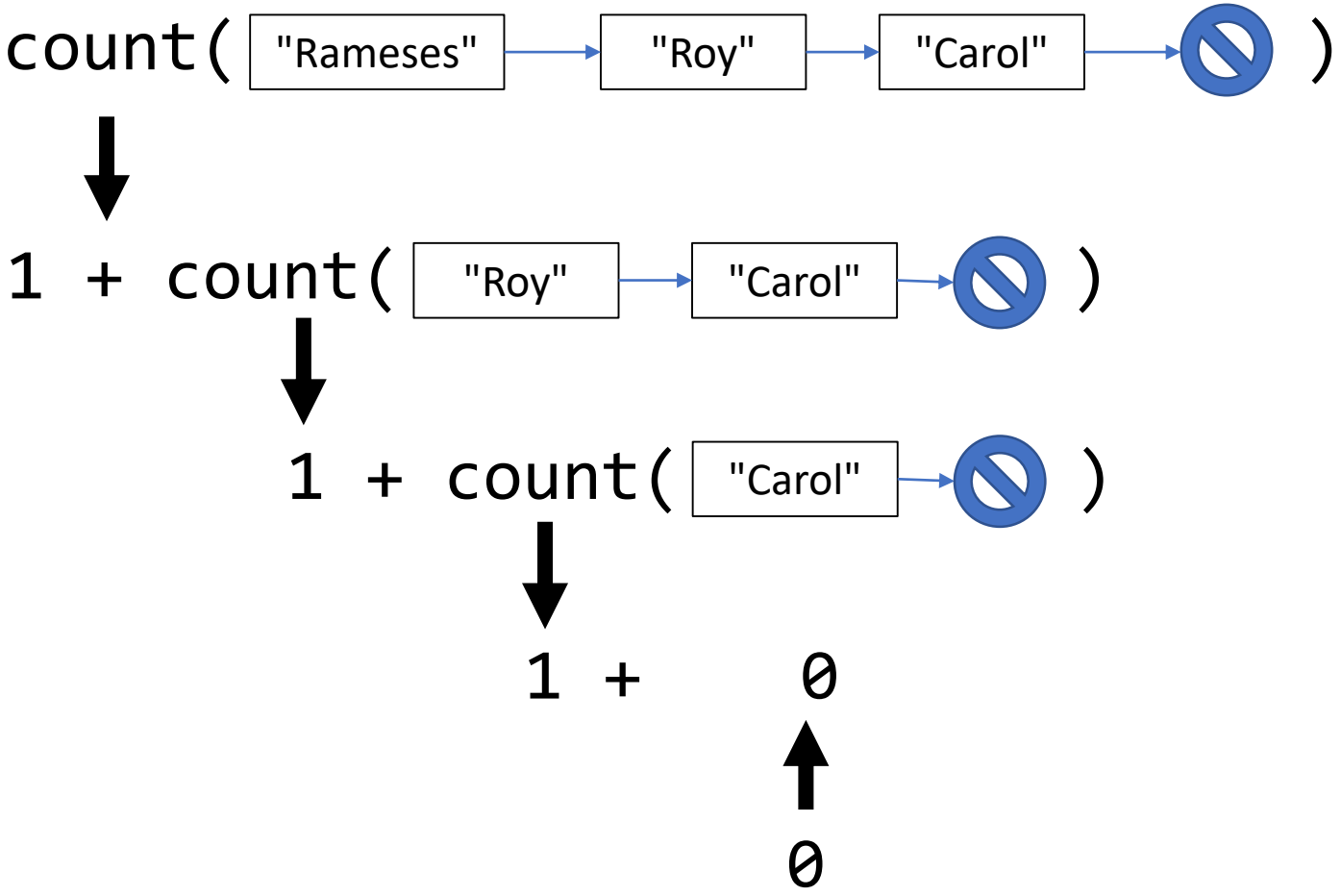
Tracing **count**

```
let count = (list: Node): number => {
    if (list === null) {
        return 0;
    } else {
        return 1 + count(rest(list));
    }
};
```



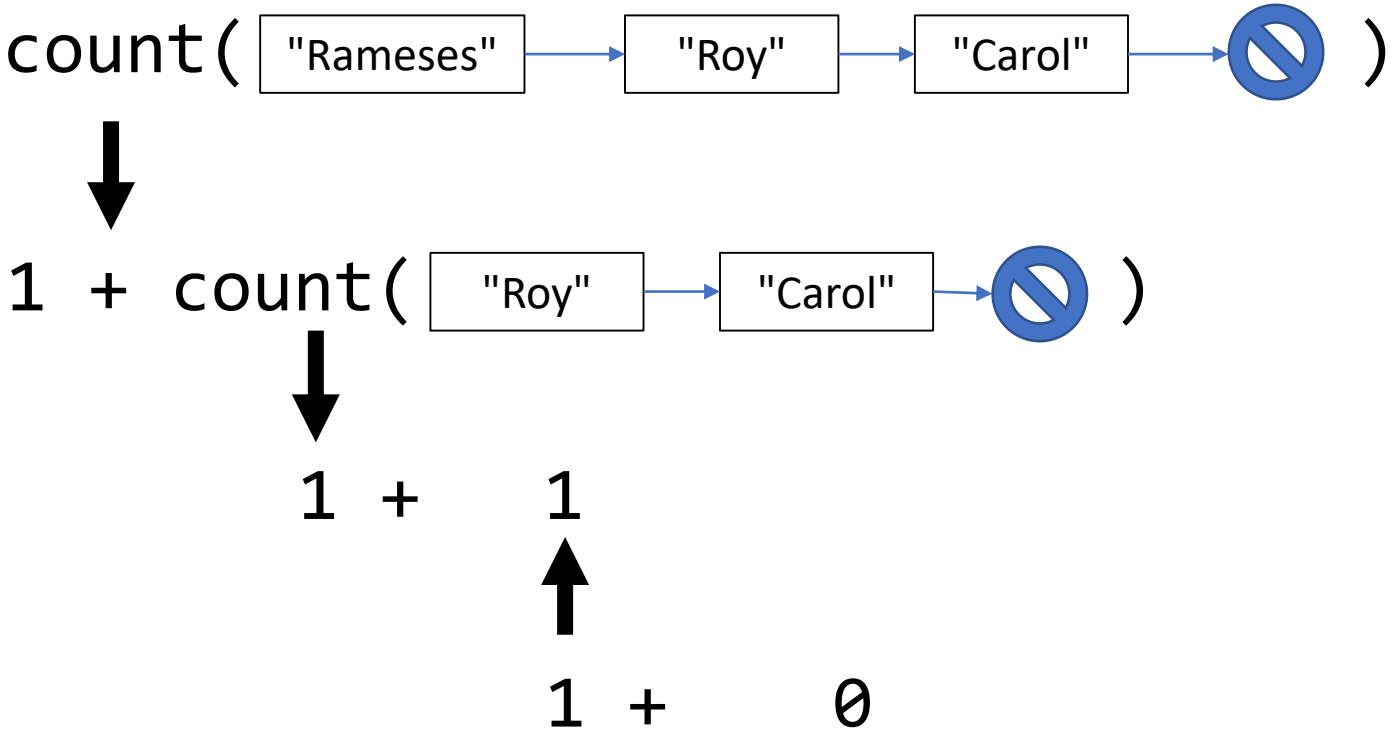
Tracing **count**

```
let count = (list: Node): number => {
    if (list === null) {
        return 0;
    } else {
        return 1 + count(rest(list));
    }
};
```



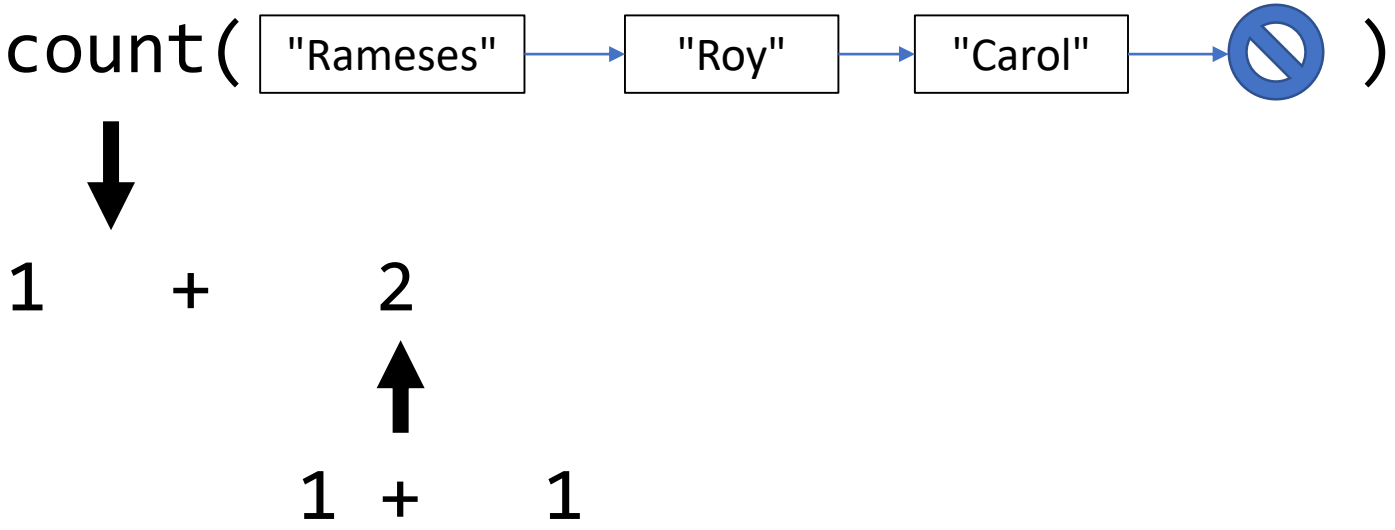
Tracing **count**

```
let count = (list: Node): number => {
    if (list === null) {
        return 0;
    } else {
        return 1 + count(rest(list));
    }
};
```



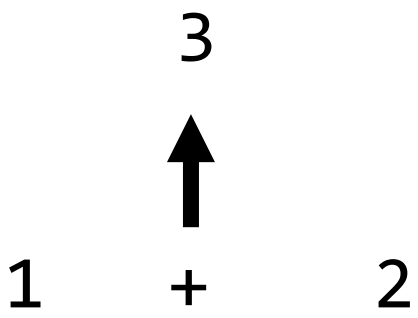
Tracing **count**

```
let count = (list: Node): number => {
    if (list === null) {
        return 0;
    } else {
        return 1 + count(rest(list));
    }
};
```



Tracing **count**

```
let count = (list: Node): number => {
    if (list === null) {
        return 0;
    } else {
        return 1 + count(rest(list));
    }
};
```



Rules of List Recursion

When using recursion to process a List:

1. Always test to see if the list is *empty* (equal to `null`)
   - This is the "base case"! *Recursion is all about that bass…*

2. Make the recursive call with the *rest of the list*

Rules of Recursion using Lists

1. Always check if list is empty! This is the base case.

```
let count = (list: Node): number => {
    if (list === null) {
        return 0;
    } else {
        return 1 + count(rest(list));
    }
};
```

2. Make the recursive call with the *rest of the list.*

Does a List ***include*** a specific value? `true/false`

- How can we write a function that, given a list of any length and a search value, we can check to see if the list contains that value?

- Let's try it with ***pseudo-code*** first!

- **Includes Algorithm**, Given any list and a value **V**
  1. If the List is empty, then the List does not include V, return false!
  2. Else,
     1. If the first value in the List equals **V**, return true! – first(list)
     2. Else, run the includes algorithm on the rest of the list – rest(list)

- UTA Live Demo

Follow-along: Includes Algorithm

- Let's open example 03 and write the following algorithm for the ***includes*** function together.

```
let includes = (list: Node, search: string): boolean => {
    if (list === null) {
        return false;
    } else if (first(list) === search) {
        return true;
    } else {
        return includes(rest(list), search);
    }
};
```

Rules of Recursion using Lists

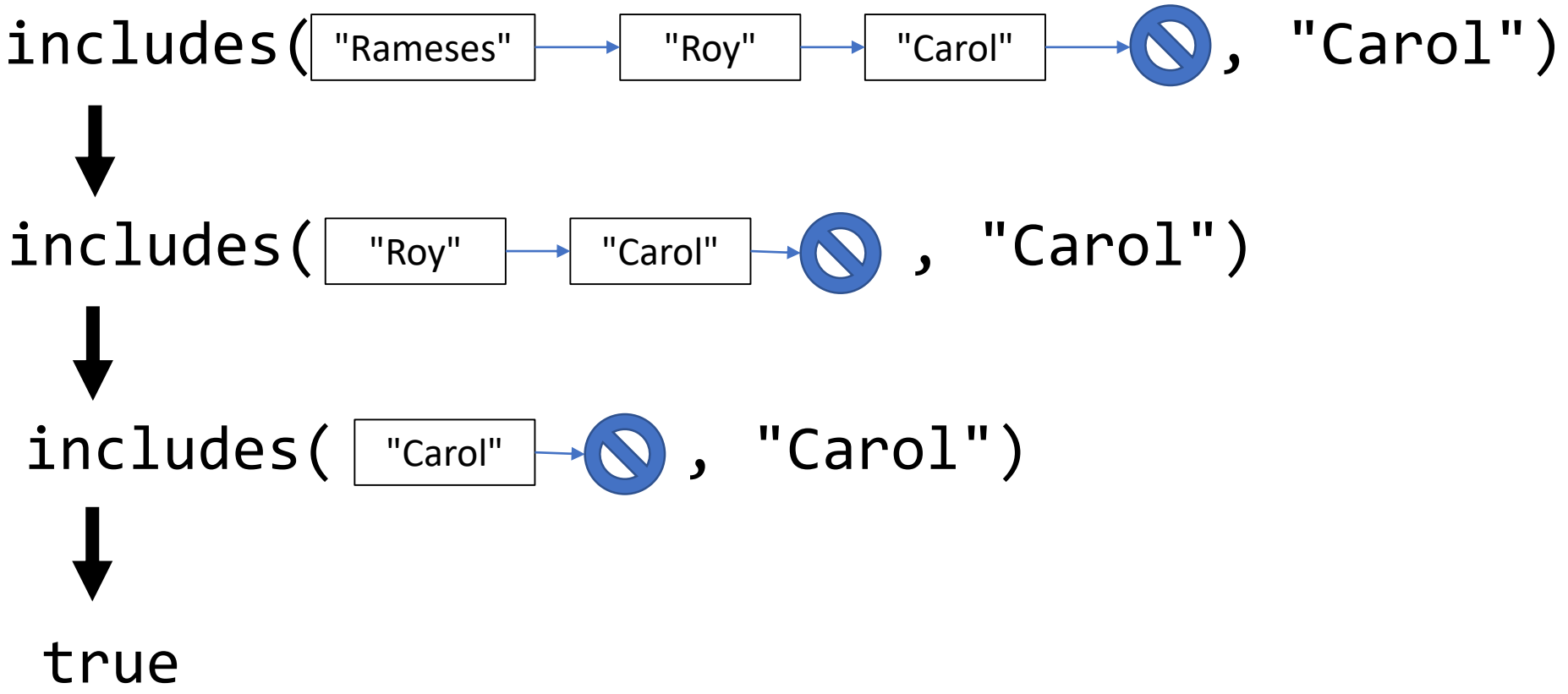
1. Always check if list is empty! This is the base case.

```
let includes = (list: Node, search: string): boolean => {
    if (list === null) {
        return false;
    } else if (first(list) === search) {
        return true;
    } else {
        return includes(rest(list), search);
    }
};
```

2. Make the recursive call with the *rest of the list.*

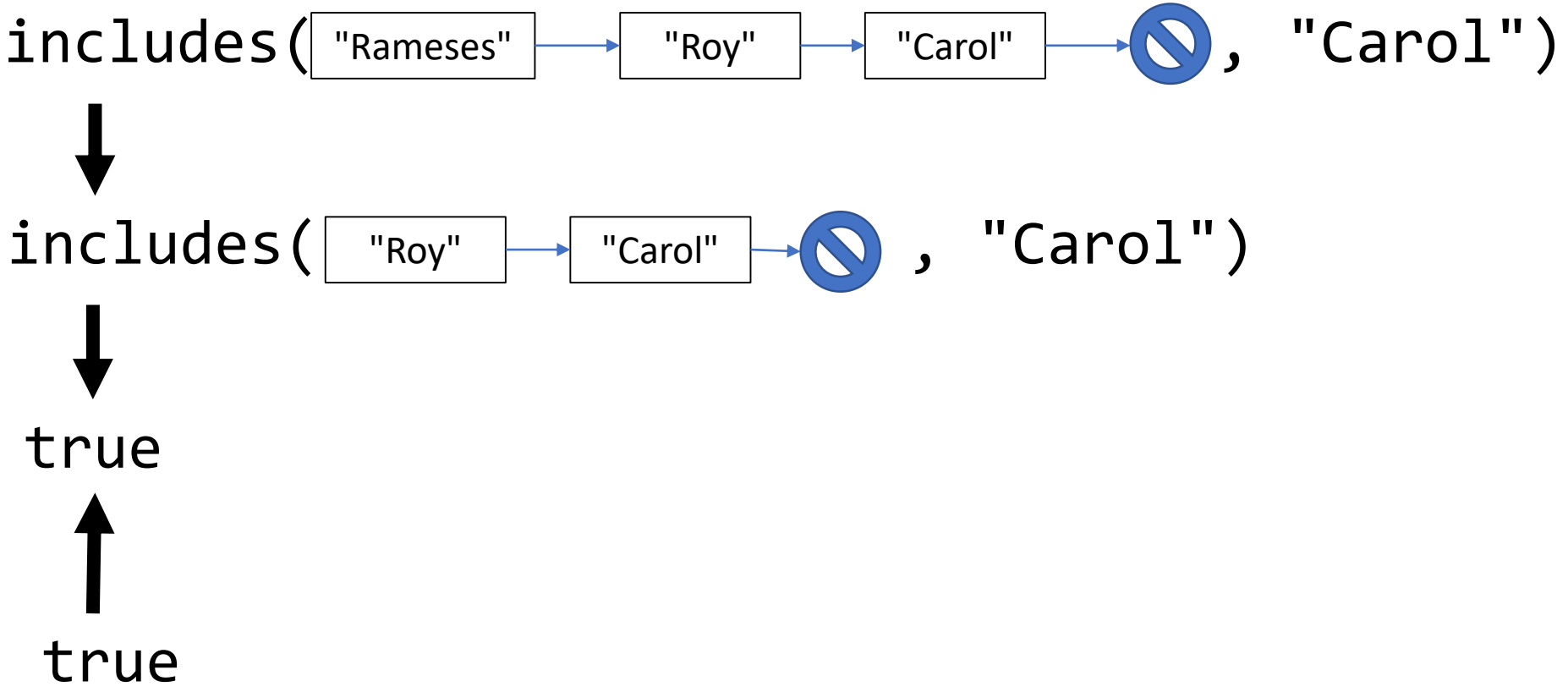
Tracing **includes**

```
let includes = (list: Node, search: string): boolean => {
     if (list === null) {
          return false;
     } else if (first(list) === search) {
          return true;
    } else {
         return includes(rest(list), search);
    }
};
```



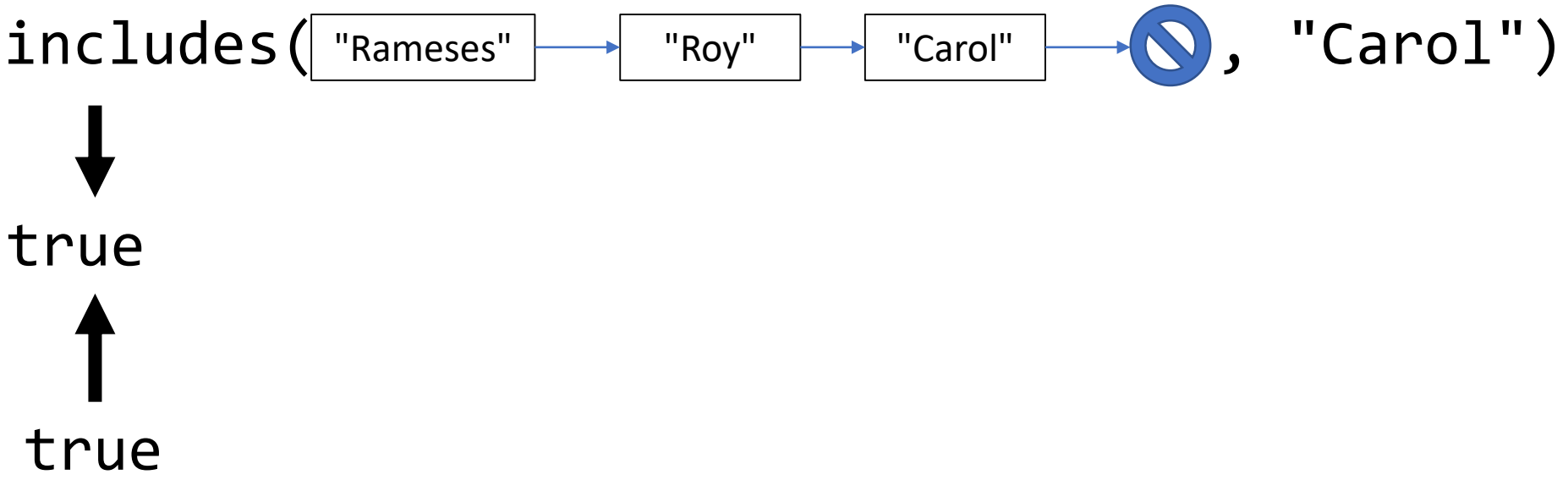
Tracing **includes**

```
let includes = (list: Node, search: string): boolean => {
     if (list === null) {
          return false;
     } else if (first(list) === search) {
          return true;
    } else {
         return includes(rest(list), search);
    }
};
```



Tracing **includes**

```
let includes = (list: Node, search: string): boolean => {
     if (list === null) {
          return false;
     } else if (first(list) === search) {
         return true;
    } else {
        return includes(rest(list), search);
    }
};
```



Tracing **`includes`**

```
let includes = (list: Node, search: string): boolean => {
     if (list === null) {
          return false;
     } else if (first(list) === search) {
         return true;
    } else {
        return includes(rest(list), search);
    }
};
```

`true`

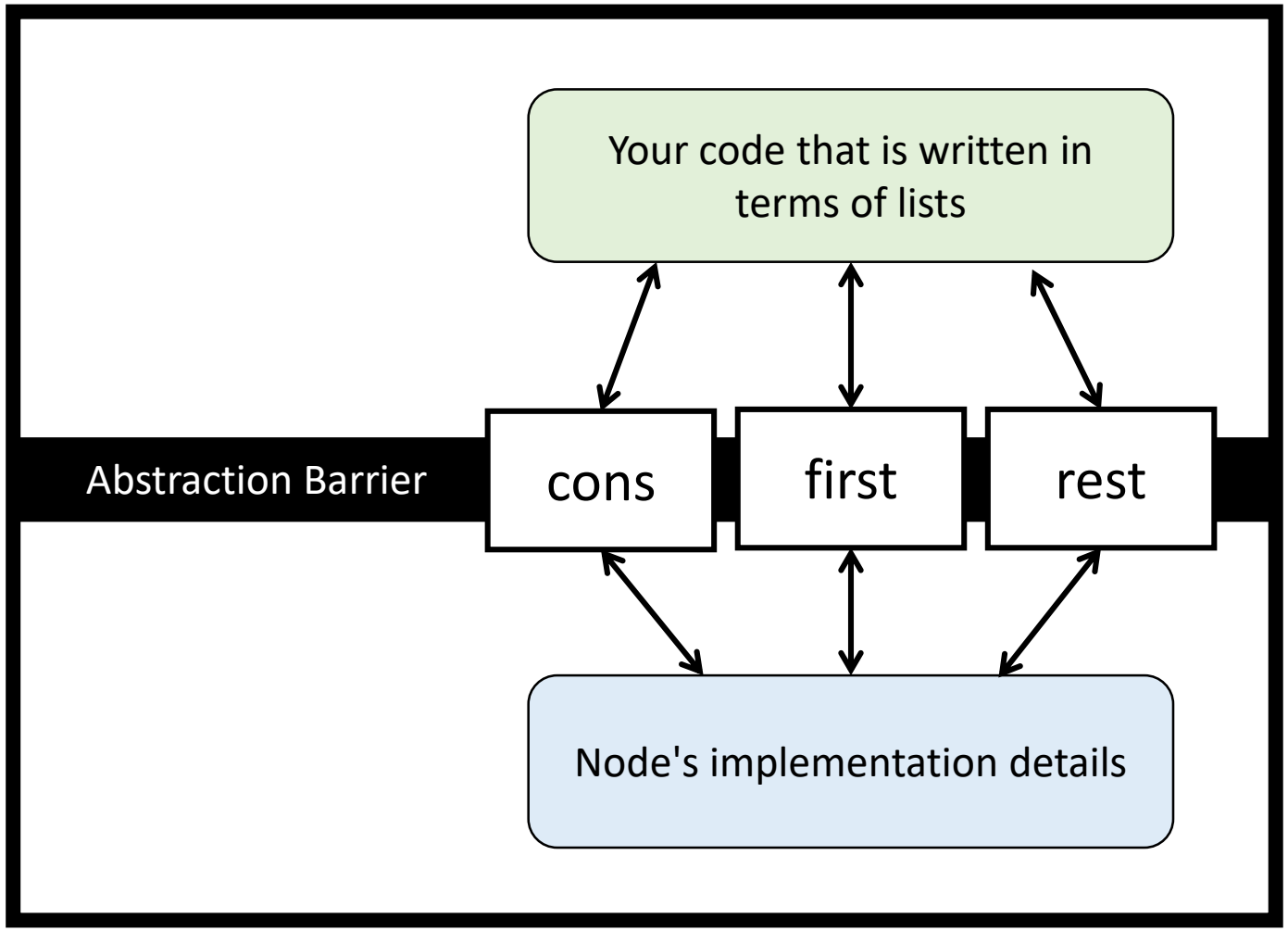
`true`

Barriers of Abstraction (1 / 2)

- To build complex systems you must manage complexity by abstracting away or "hiding" implementation details

- **Abstraction barriers** are common across all engineering disciplines
  - Fundamental to organizational management, too!

- Code on one side of an abstraction barrier knows nothing about what's on the other side.



Barriers of Abstraction (2 / 2)

- In our example, the **abstraction barrier** are the `cons`, `first`, `rest`.

- By writing code that *only* depends on these three functions we:

1. Simplify our code and avoid having to do all the bookkeeping.

2. Make it possible to change or improve code on either side of the abstraction barrier *independent of the other side*.

- Thus, we've solved our two problems with the opening example! In large scale software projects you will find many *layers of abstraction* each separated by a barrier.