Recursion on Lists and Generics

Lecture 16

Have PollEv.com/compunc ready to go as well as your page of notes!

Warm-up Questions

1. Recursive concepts are at play when...
    1. a concept is defined in terms of itself
    2. a function calls itself from within itself
    3. a class has a property of the same type as itself
2. When the call stack reaches its limit of frames...
    1. stack overflow
3. When does a recursive function stop making recursive calls?
    1. at a base case
4. To avoid infinite recursion, a recursive function call
    1. call itself with different arguments than it was called with, make progress toward the base case
5. The first node of a linked list is conventionally called
    1. the head node
6. You can think of null as
    1. A reference to nowhere
    2. The end of a linked list
7. The cons function is used to
    1. construct a list
8. In Foo -> Bar -> Baz -> null -- you would access Bar with
    1. first(rest(list))

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

```
1  let odd = (n: number): number => {
2    if (n % 2 === 1 && n % 3 === 0) {
3      return n;
4    } else {
5      return odd(n + 1);
6    }
7  };
```

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

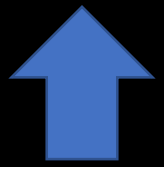
- UTA Live Demo

Follow-along: Implementing Count in Code

- Let's open 00-count-app.ts and implement the count function together

count in code

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

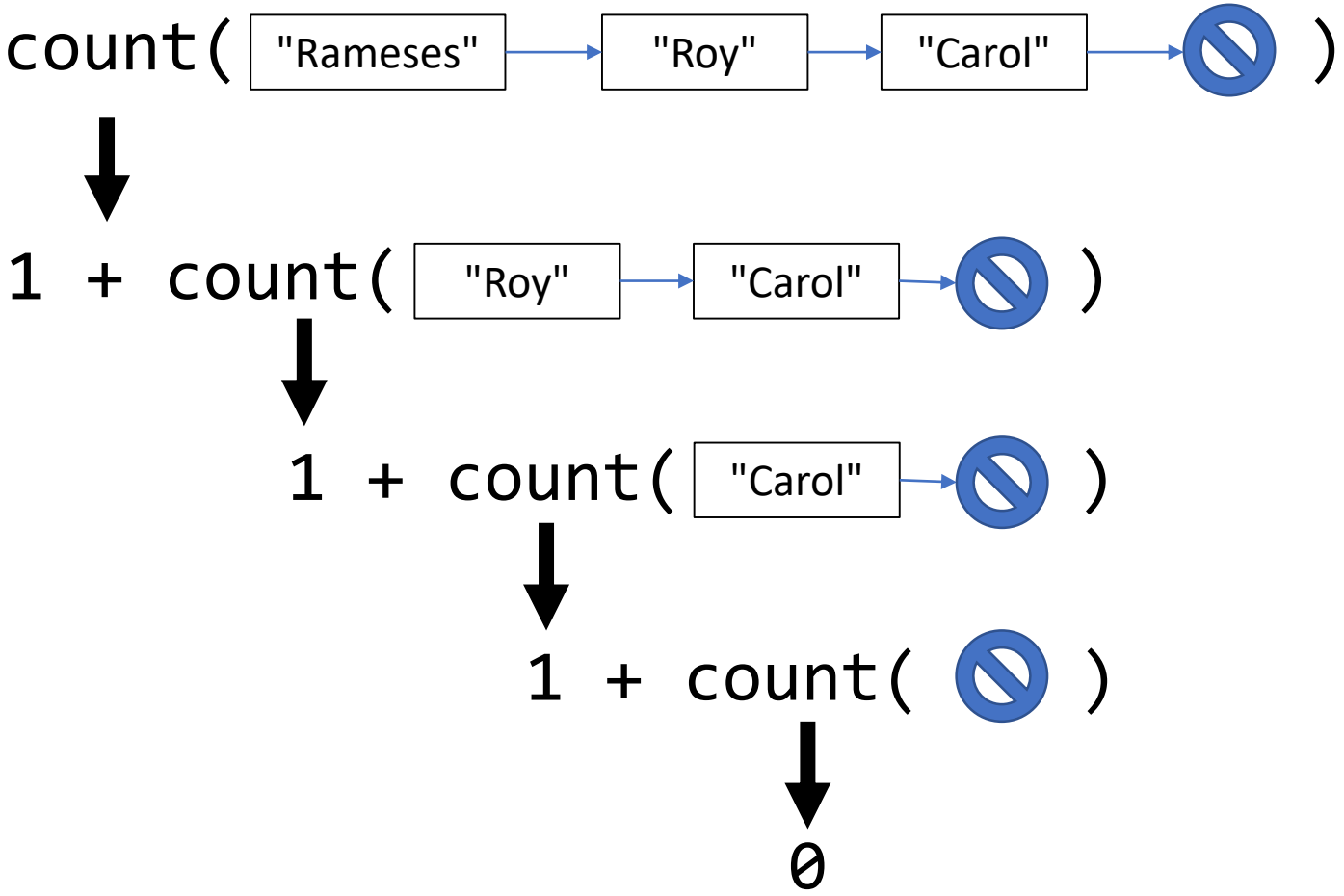


- What magic is this?

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

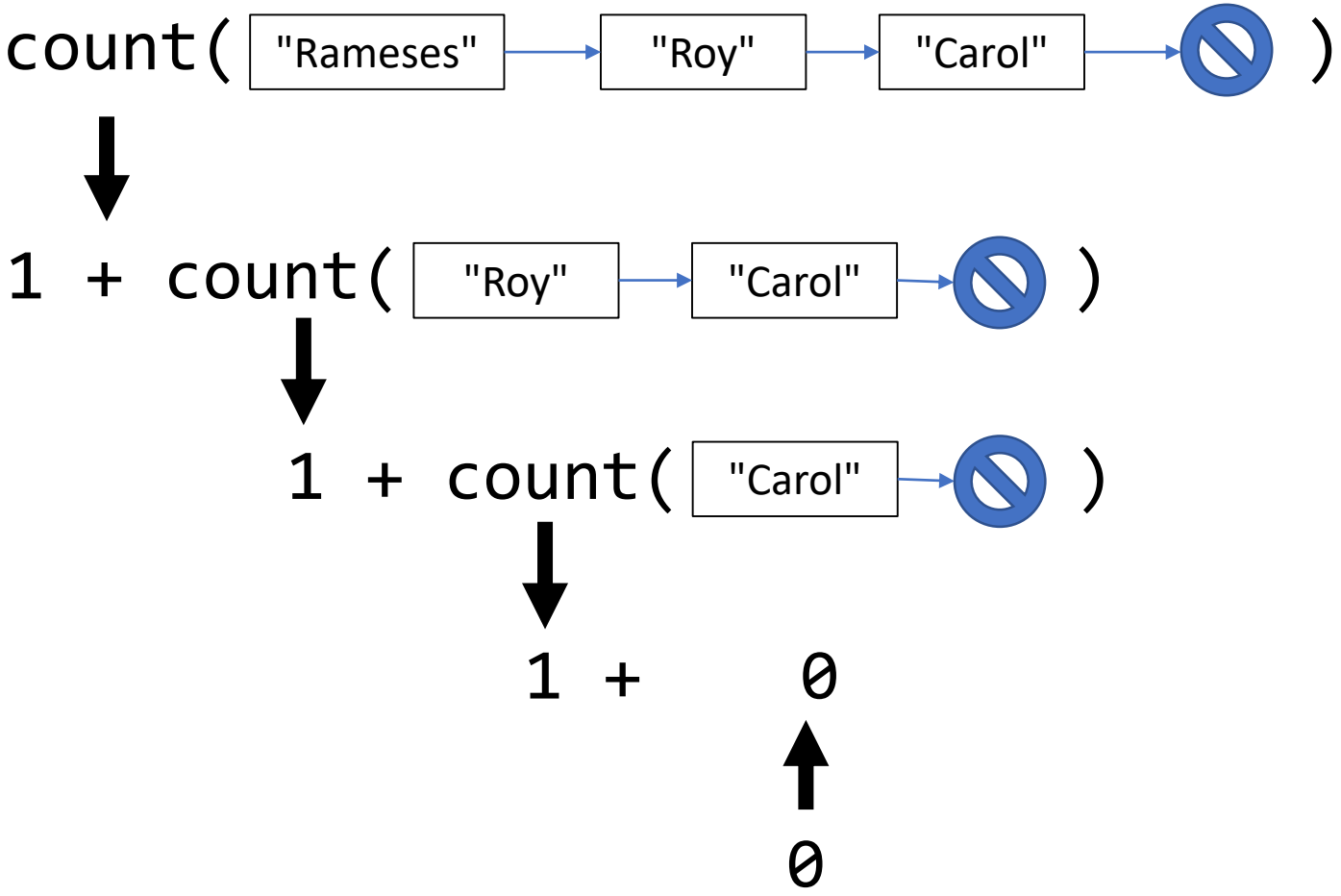
Tracing **count**

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



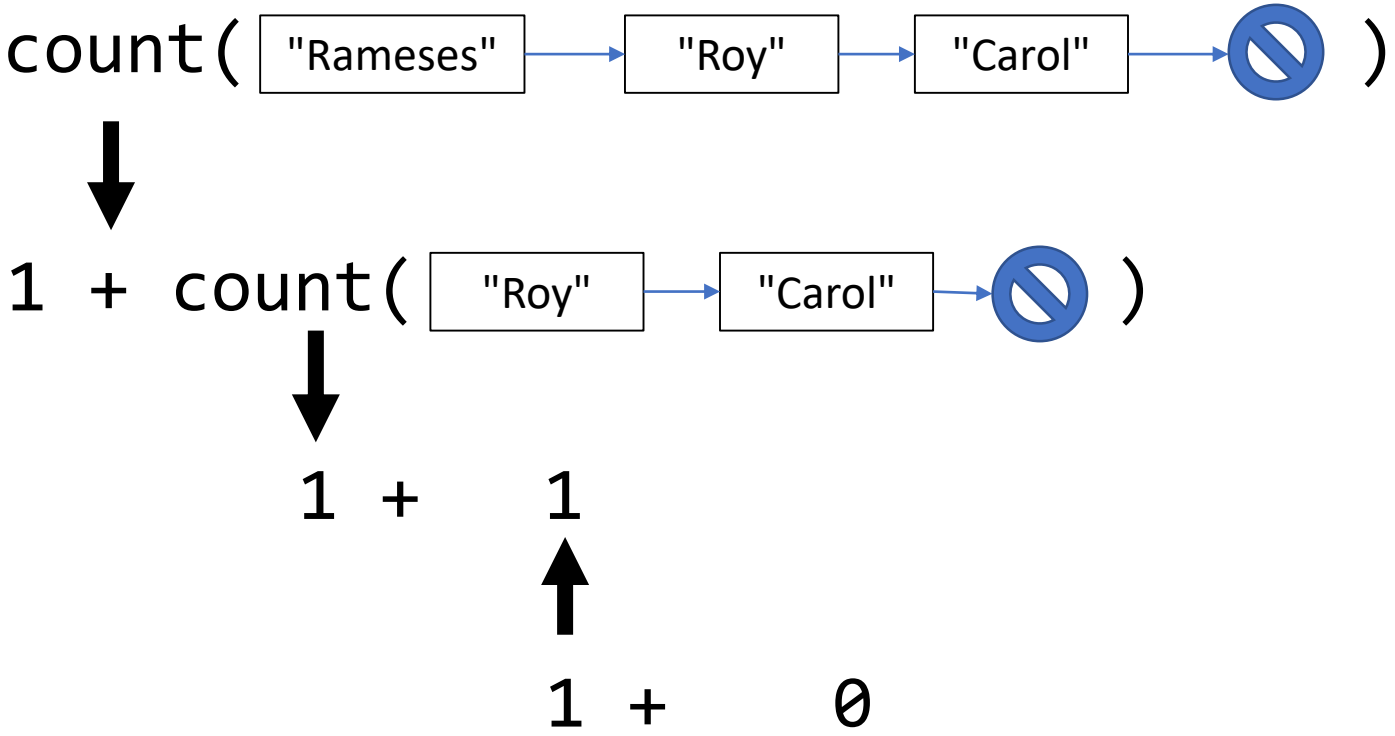
Tracing **count**

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



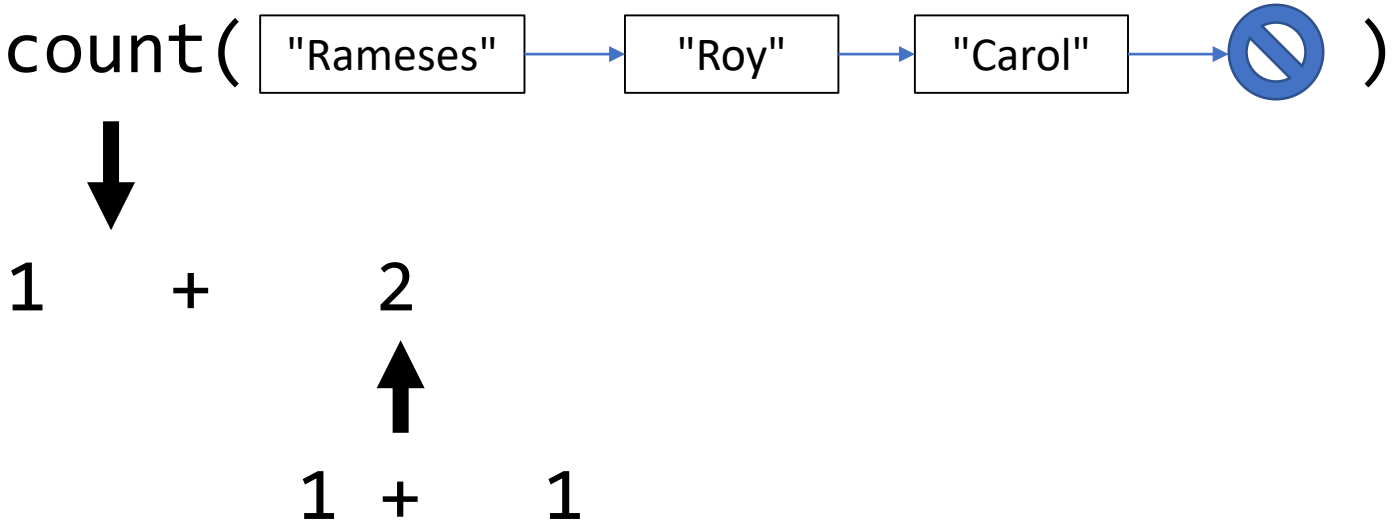
Tracing **count**

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



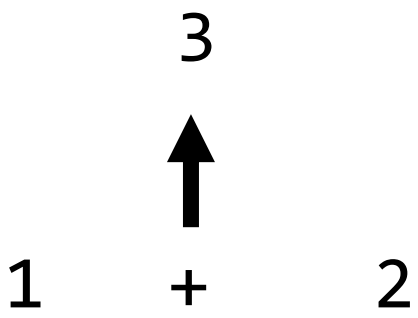
Tracing **count**

```
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 01 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

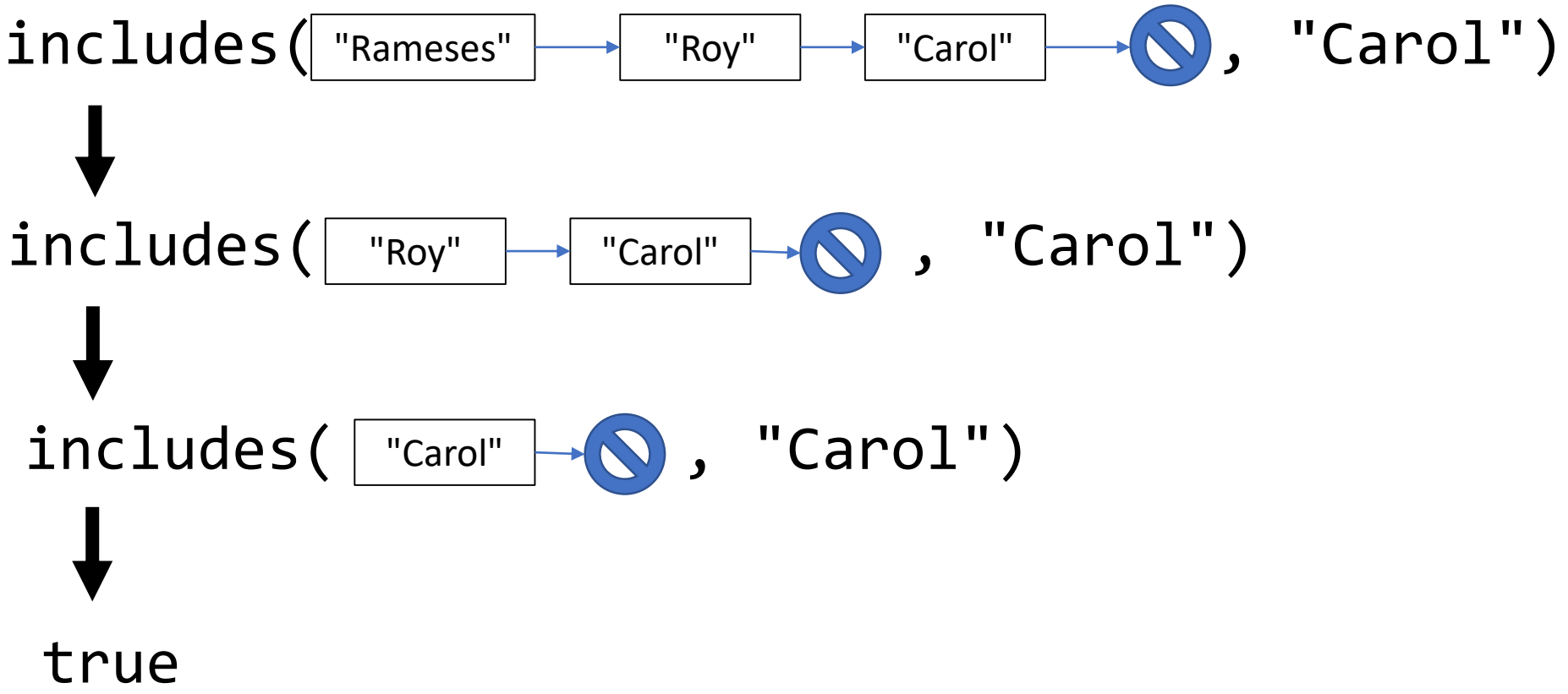
1. Always check if list is empty! This is the base case.

```
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.*

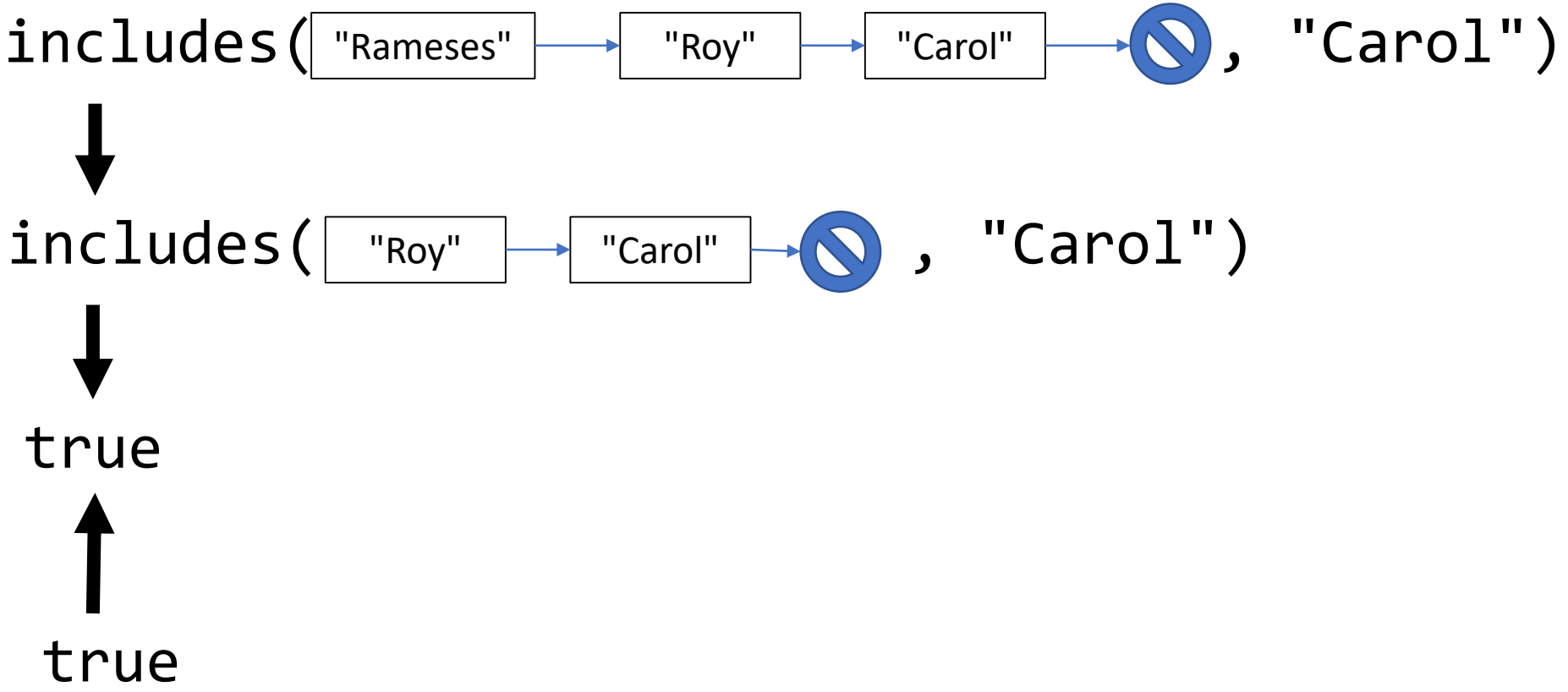
Tracing **includes**

```
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);
    }
};
```



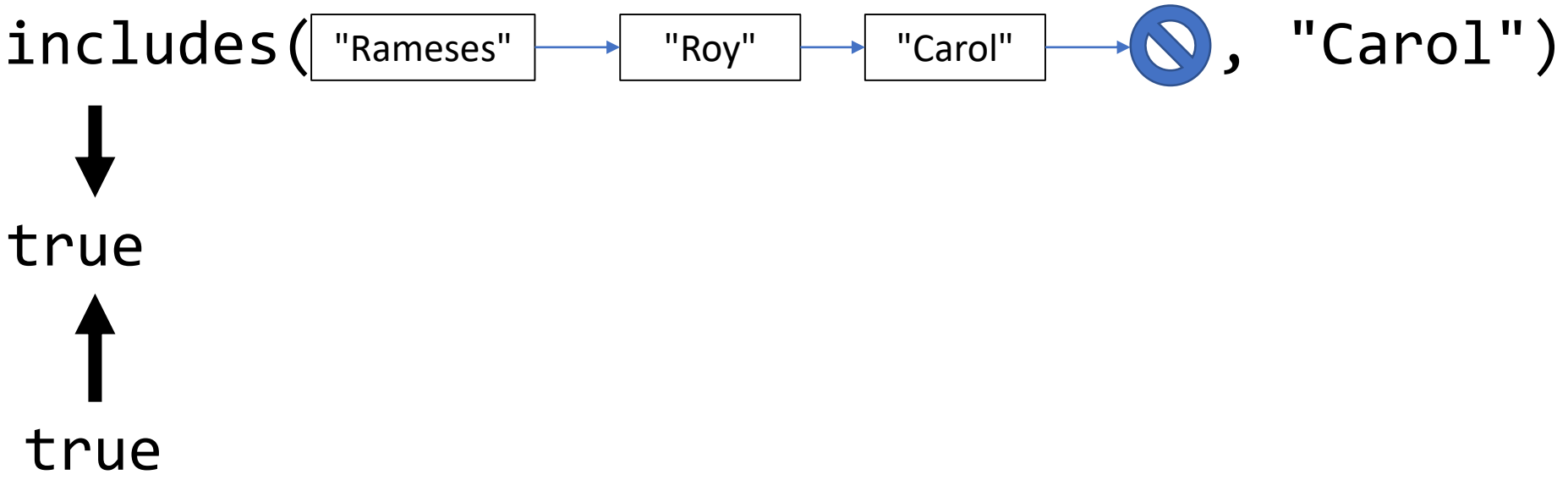
Tracing **includes**

```
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`**

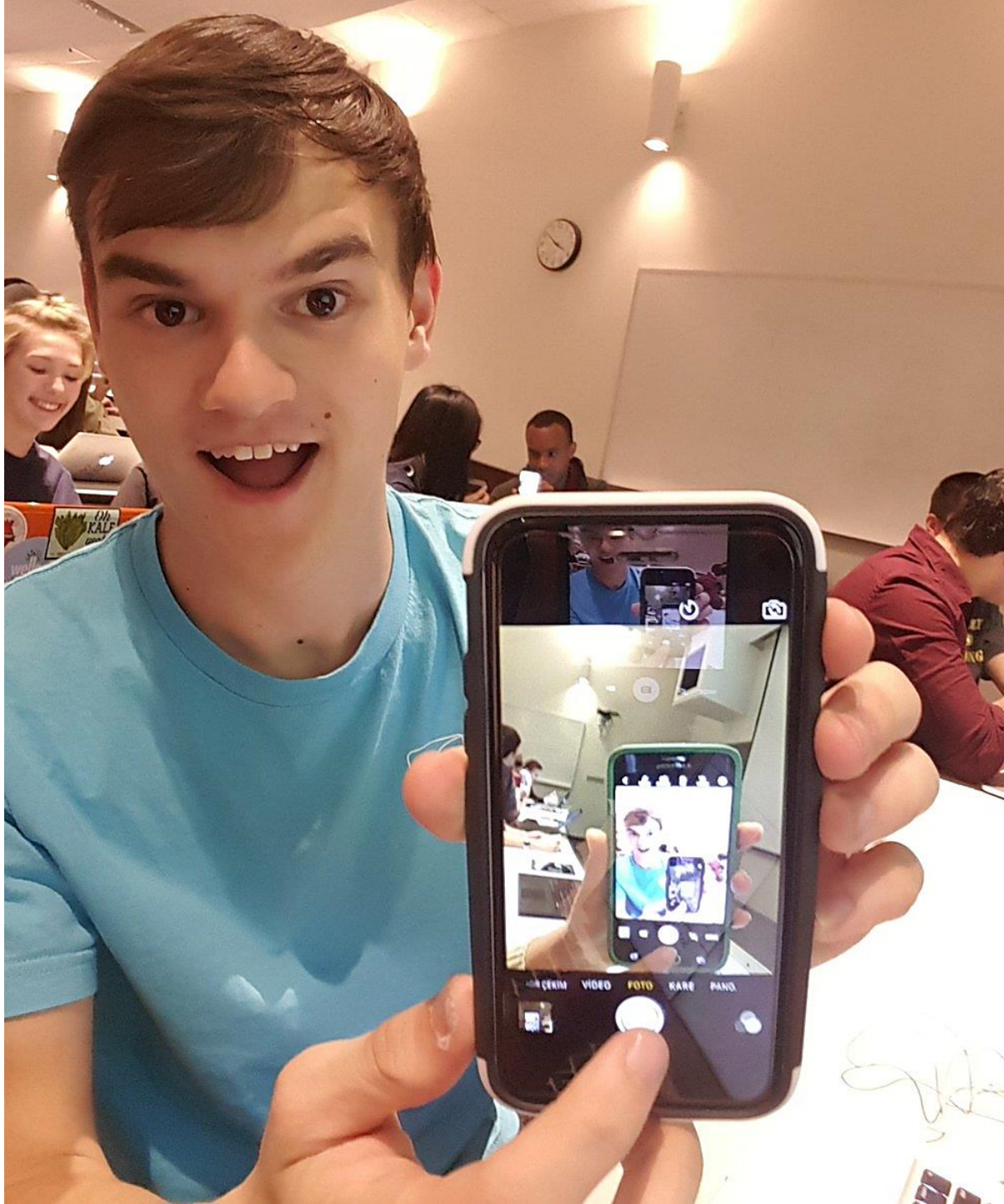
```
let includes = (list: Node, search: string): boolean => {
     if (list === null) {
          return false;
     } else if (first(list) === search) {
         return true;
    } else {
        return includes(rest(list), search);
    }
};
```

`true`

`true`

Recursive Selfie

- Pair up with a neighbor!
- Turn on your front facing cameras
- Hold your cameras between you and point the front of your phones at each other…
- Now try to get in the picture!
- Post to Twitter with #comp110
- Check-in on PollEv.com/compunc



Generic Types

- When working with **collections** of data, like lists and arrays, you often want capabilities (functions/algorithms) that work regardless of the collection's data type
  - i.e. does a list of strings include a specific string? does a list of numbers include a specific number?

- Creating a data structure class and related functions per data type leads to a *lot* of repetition in code.

- **Generic types** offer a solution and enable you to parameterize your data types at a class or function level.

Generic Classes

- When 2 classes differ only by the types of their properties you should use a generic class instead.

```
class NodeNumber {
   data: number = 0;
   next: NodeNumber = null;
}
```

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

Generic Classes

- **Step 1:** Designate a class as being generically typed and update any recursively typed properties

- Place a "diamond" **<>**, with a name placeholder in it like **<T>**, after the class name:

```
class Node<T> {
   data: number = 0;
   next: Node<T> = null;
}
```

- You can read this as "Class Node is generic for any type T"
- The use of the capital letter **T**(ype), is only a convention. We could place another letter, like **U**, or even a word here, like **TYPE .**

Generic Classes have Generic Properties

- **Step 2:** Change the relevant properties to be generically typed.

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

- Once a generic class is defined, you can use concrete types such as:
  - `Node<number>` - where T is number, thus the `data` property's type is number
  - `Node<string>` - where T is string, thus the `data` property's type is string
- **Big idea**: using only one generic class definition for Node, you can work with Node objects that hold data of any type!
- Note: you cannot assign a default value to a generic property. Think about why not.

Follow-Along: Define a generic Node class.

- In 02-generic-class-app.ts, let's define the following class together:

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

Generic Classes - Constructing Objects

- **Step 3:** Constructing objects of generic types.

```
// Explicit Typing
let a: Node<string> = new Node<string>();
a.data = "hello, world";

// Type Inference
let b = new Node<number>();
b.data = 110;
```

You can use the concrete types anywhere you could otherwise use a class name. For example, declaring variables and constructing objects.

Note, however, the concrete types Node<string> and Node<number> are not the same type! For example, trying to assign `a.next = b;` in the code above will error.

What's different about these two functions?

```
export let consString = (data: string, next: Node<string>): Node<string> => {
    let n = new Node<string>();
    n.data = data;
    n.next = next;
    return n;
};

export let consNumber = (data: number, next: Node<number>): Node<number> => {
    let n = new Node<number>();
    n.data = data;
    n.next = next;
    return n;
};
```

Generic Functions (1 / 4)

- Do we *really* need to duplicate the logic of functions like **cons** for every concrete type of **Node**?

- Good news! No, we do not thanks to **generically typed functions**.

- Rule of Thumb: When 2+ functions differ *only* in parameter types or return type, you can replace them with a single generic function.

Generic Functions (2 / 4)

- **Step 1:** Designate a function as a function that is "Generic for any Type"

- Place a "diamond" **<>**, with a name in it like **<T>**, before the parameter list:

```
let cons = <T> (data: string, next: Node<string>): Node<string> => {
    // ...
};
```

- As with classes, the use of the capital letter **T**(ype), is only a convention. Additionally, the choice of using T in the class definition and then again here are independent decisions. It's good style for generic functions to use the same generic type name conventions as the generic classes they operate on, though.

Generic Functions (3 / 4)

- **Step 2:** Identify the types that changed between your otherwise identical functions.

```
let consString = (data: string, next: Node<string>): Node<string> => {
```

```
let consNumber = (data: number, next: Node<number>): Node<number> => {
```

- Replace the types that changed with the generic type T:

```
let cons = <T> (data: T, next: Node<T>): Node<T> => {
```

- Finally, inside the function definition body, replace specific type with generic.

Generic Functions (4 / 4)

- **Step 3:** Inside the function definition body, replace specific types with generic.

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

- The programming language will catch the type error above until corrected.

```
export let cons = <T> (data: T, next: Node<T>): Node<T> => {
    let n = new Node<T>();
    n.data = data;
    n.next = next;
    return n;
};
```

Hands-on: Making a Generic **cons** Function

- Open lec16 / 02-generic-functions-app.ts

1. Convert the **cons** function to be a generic function.
   a) Add the diamond T syntax before the parameter list: **let cons = <T> (...**
   b) Replace the specific **string** type with the generic type **T**
      - **string** is replaced with **T**
      - **Node<string>** is replaced with **Node<T>**

2. In the main function, change the specific calls to consString and consNumber to use the generic cons function.

3. Remove the consString and consNumber functions from the program.

- Check-in on PollEv.com/compunc when your generic `cons` function is working

```
export let cons = <T> (data: T, next: Node<T>): Node<T> => {
   let n = new Node<T>();
   n.data = data;
   n.next = next;
   return n;
};
```

Building Lists Recursively

- Previously, our recursive functions took a list as a parameter and then recursively computed *a single value*

- Can we write a function that takes a List as a parameter and then *builds and returns another List*?

- To build a List, we'll process the first value of the List and *cons* it onto the result of processing the rest of the list recursively
  - So far, our recursive functions have processed the `first` value of a List
  - Now we'll use the *cons* function to add a value on to the front of a new List

Tracing **acronymify**

```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```

acronymify("Michael" → "Jordan" → null);

Tracing **acronymify**

```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```

```
acronymify( "Michael" →  "Jordan" →  null );
          ⬇
     cons( "M",  acronymify( "Jordan" →  null ) );
```

To build a list, process the first value, and then cons it onto the result of repeating the same process recursively on the rest of the list.

Tracing **acronymify**

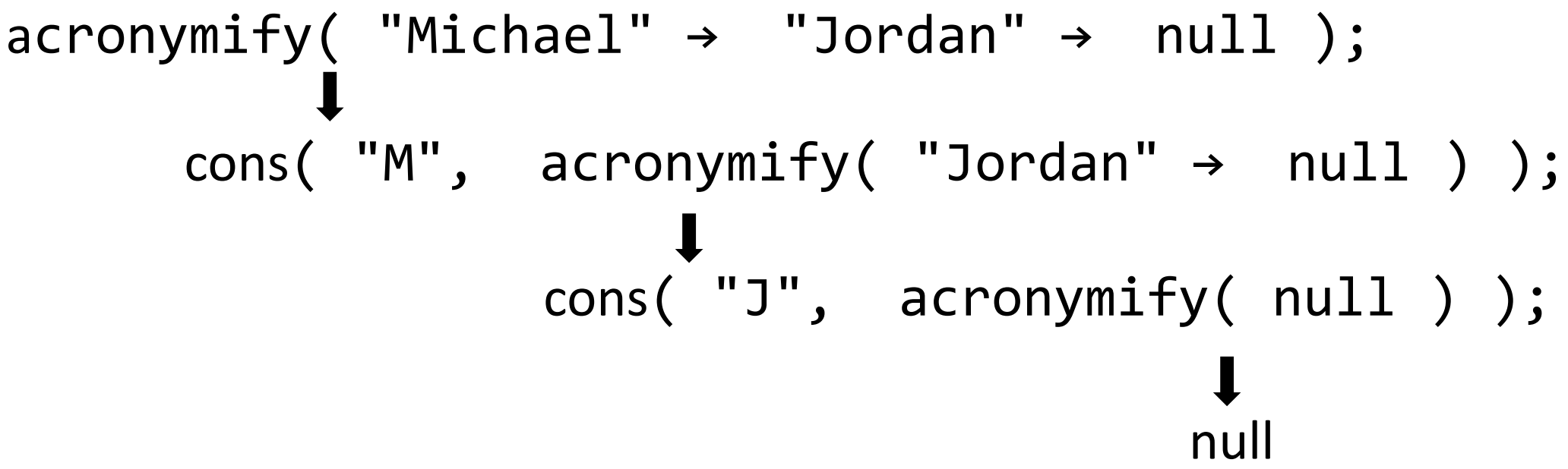
```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```

```
acronymify( "Michael" → "Jordan" → null );
          ⬇
    cons( "M", acronymify( "Jordan" → null ) );
                        ⬇
                 cons( "J", acronymify( null ) );
```

To build a list, process the first value, and then cons it onto the result of repeating the same process recursively on the rest of the list.

Tracing **acronymify**

```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```



Tracing **acronymify**

```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```

```
acronymify( "Michael" →  "Jordan" →  null );
          ⬇
     cons( "M",  acronymify( "Jordan" →  null ) );
                       ⬇
                 cons( "J",  null );
                       ⬆
                     null
```

Tracing **acronymify**

```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```

```
acronymify( "Michael" →  "Jordan" →  null );
          ⬇
      cons( "M",  cons( "J", null ) );
                        ⬆
                  cons( "J",  null );
```

Tracing **acronymify**

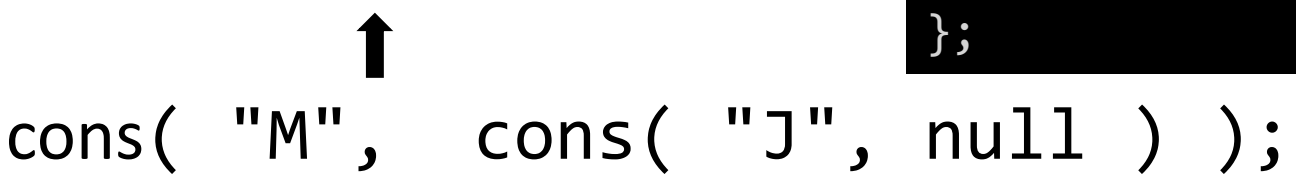
```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```

```
cons( "M",  cons( "J", null ) );
```

```
    cons( "M",  cons( "J", null ) );
```

Tracing **acronymify**

```
let acronymify = (list: Node<string>): Node<string> => {
    if (list === null) {
        return null;
    } else {
        let word = first(list);
        let letter = word[0];
        return cons(letter, acronymify(rest(list)));
    }
};
```




```
cons( "M",  cons( "J", null ) );
```

Rules of Recursive Functions

1. Test for a base case (empty list)

2. Always change at least one argument when recurring (rest of list)

3. **To build a list, process the first value, and then cons it onto the result of repeating the same process recursively on the rest of the list**