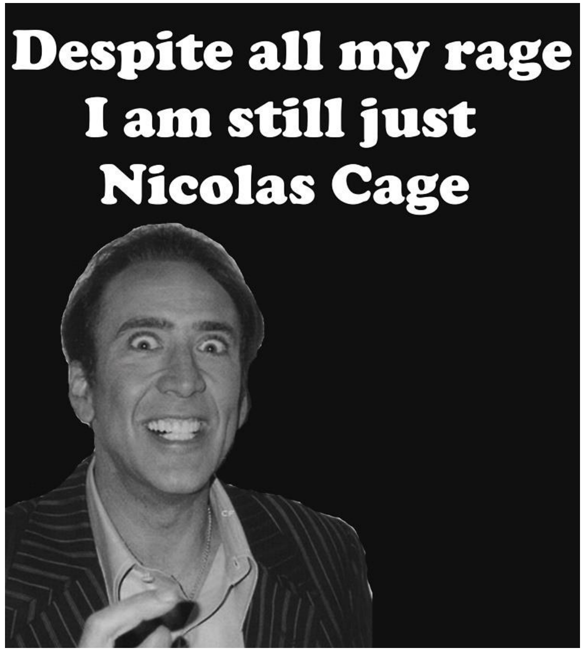
Generic Functions, else-if syntax, and Unions

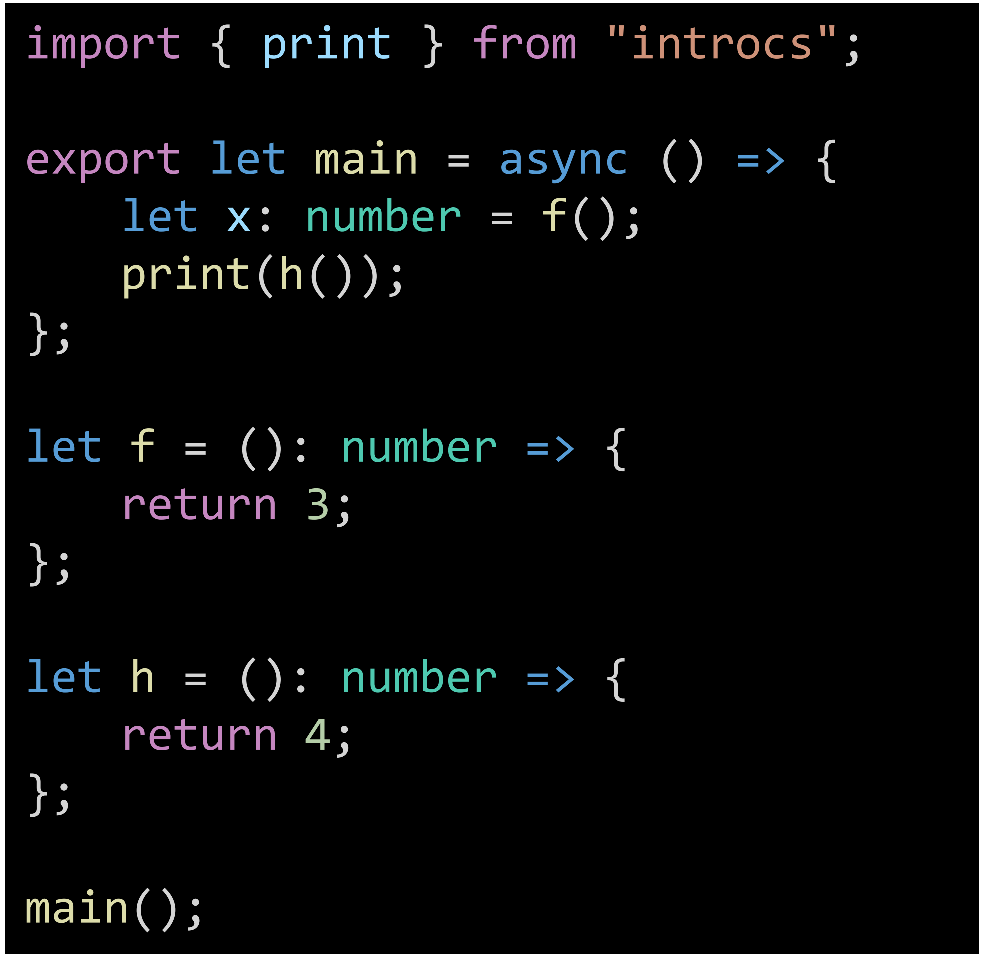
Lecture 10 - Spring 2018



Announcements

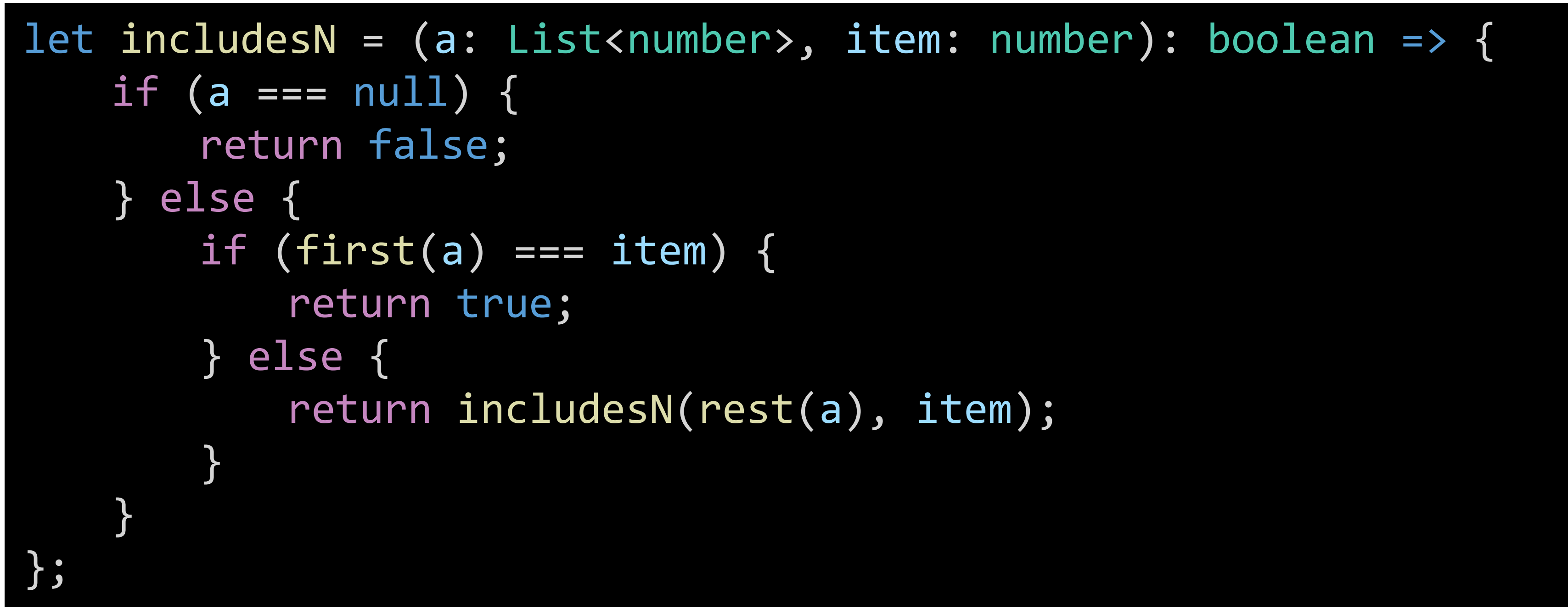
- PS02 - Mystery on the Hogwarts Express
    - Due Sunday 2/25 at 11:59pm
- This Thursday's Lecture (2/20)
    - Digital Review Session
        - Review Videos will be distributed Thursday covering midterm topics
    - Office Hours will be fully staffed during class periods
        - Come in for conceptual help and/or questions regarding PS2
    - We will not meet in Hanes Art Center
- Midterm 0 – Tuesday 2/29 - Next week!
- Review worksheet out today. This WS will not be handed-in. The answer key will post by Saturday.
- Additional Review Sessions to Prepare for MT0
    - Tomorrow at 5pm in SN014
    - Sunday at 3pm in SN014

0. What will display on the screen after this program runs?

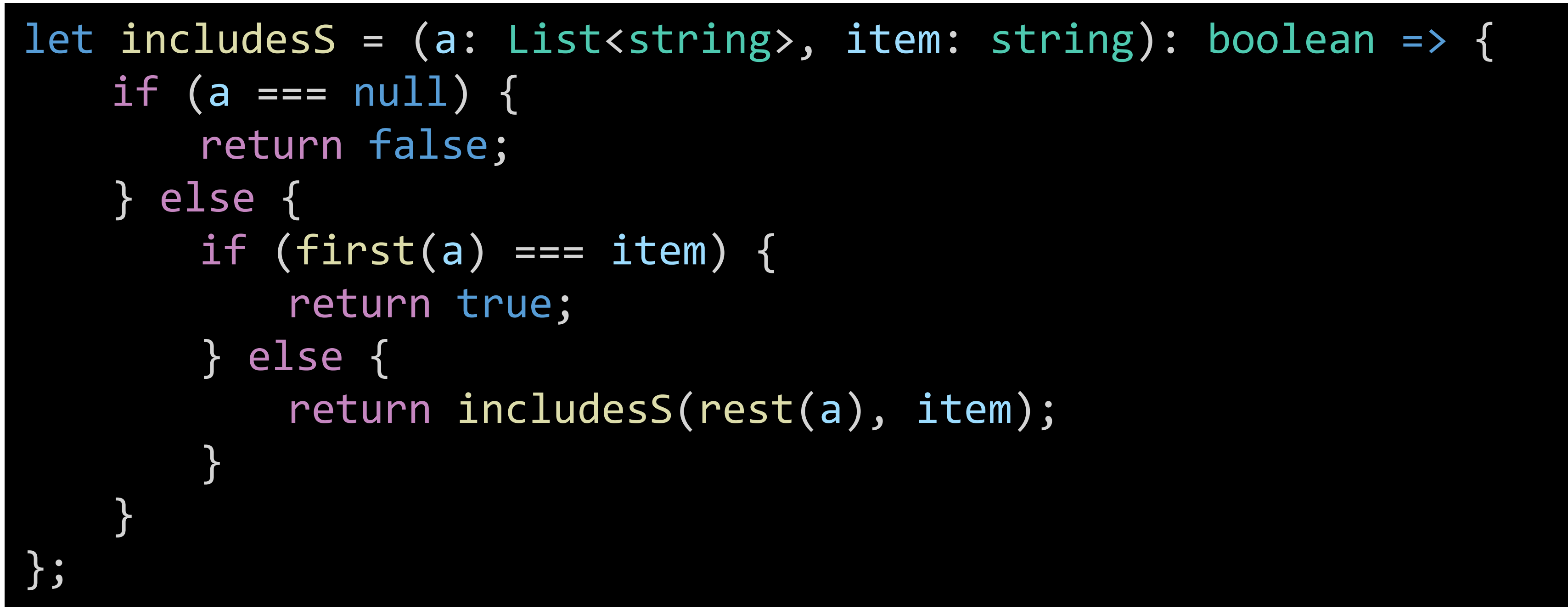


```
import { print } from "introcs";

export let main = async () => {
    let x: number = f();
    print(h());
};

let f = (): number => {
    return 3;
};

let h = (): number => {
    return 4;
};

main();
```

1. Besides their names, what are the differences between these two functions?



```
let includesN = (a: List<number>, item: number): boolean => {
    if (a === null) {
        return false;
    } else {
        if (first(a) === item) {
            return true;
        } else {
            return includesN(rest(a), item);
        }
    }
};
```



```
let includesS = (a: List<string>, item: string): boolean => {
    if (a === null) {
        return false;
    } else {
        if (first(a) === item) {
            return true;
        } else {
            return includesS(rest(a), item);
        }
    }
};
```

Generic Functions (1 / 4)

- Do we *really* need to duplicate the logic of functions like **`includes`** for every different type of **`List`**?

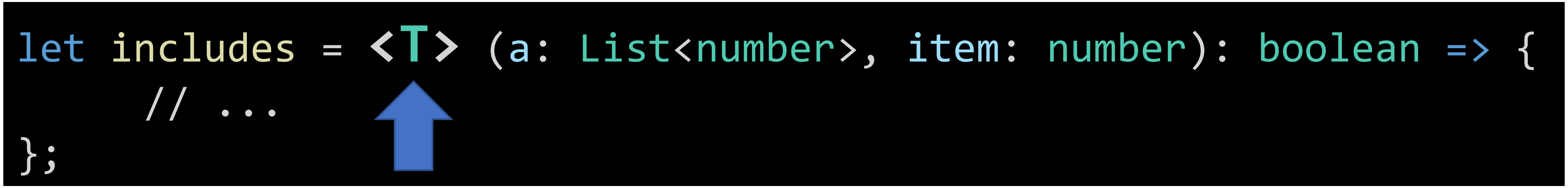
- Good news! No, we do not thanks to **generic functions**.

- 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)

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

- Place a "diamond" **<>**, with a T in it **<T>**, in front of the parameter list:



```
let includes = <T> (a: List<number>, item: number): boolean => {
     // ...
};
```

- The use of the capital letter **T**(ype), is only a convention. We could place another letter or even a word here, like TYPE.

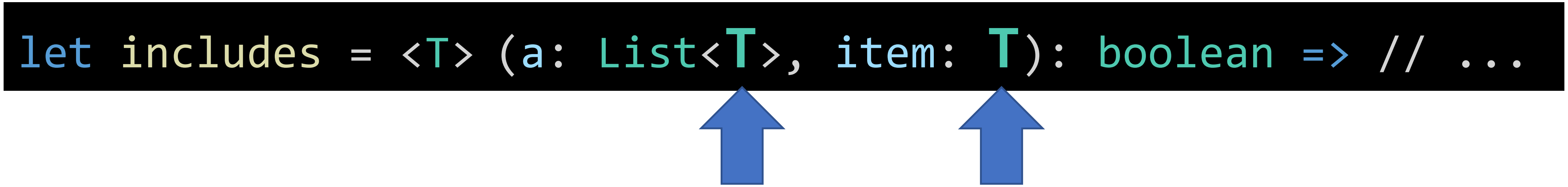
Generic Functions (3 / 4)

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

```
let includesN = (a: List<number>, item: number): boolean => // ...
```

```
let includesS = (a: List<string>, item: string): boolean => // ...
```

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



```
let includes = <T> (a: List<T>, item: T): boolean => // ...
```

- Read as "for **any type T**, if you give the **`includes`** function any `List` of T values and an item T, it will return whether the `List` includes the `item`."

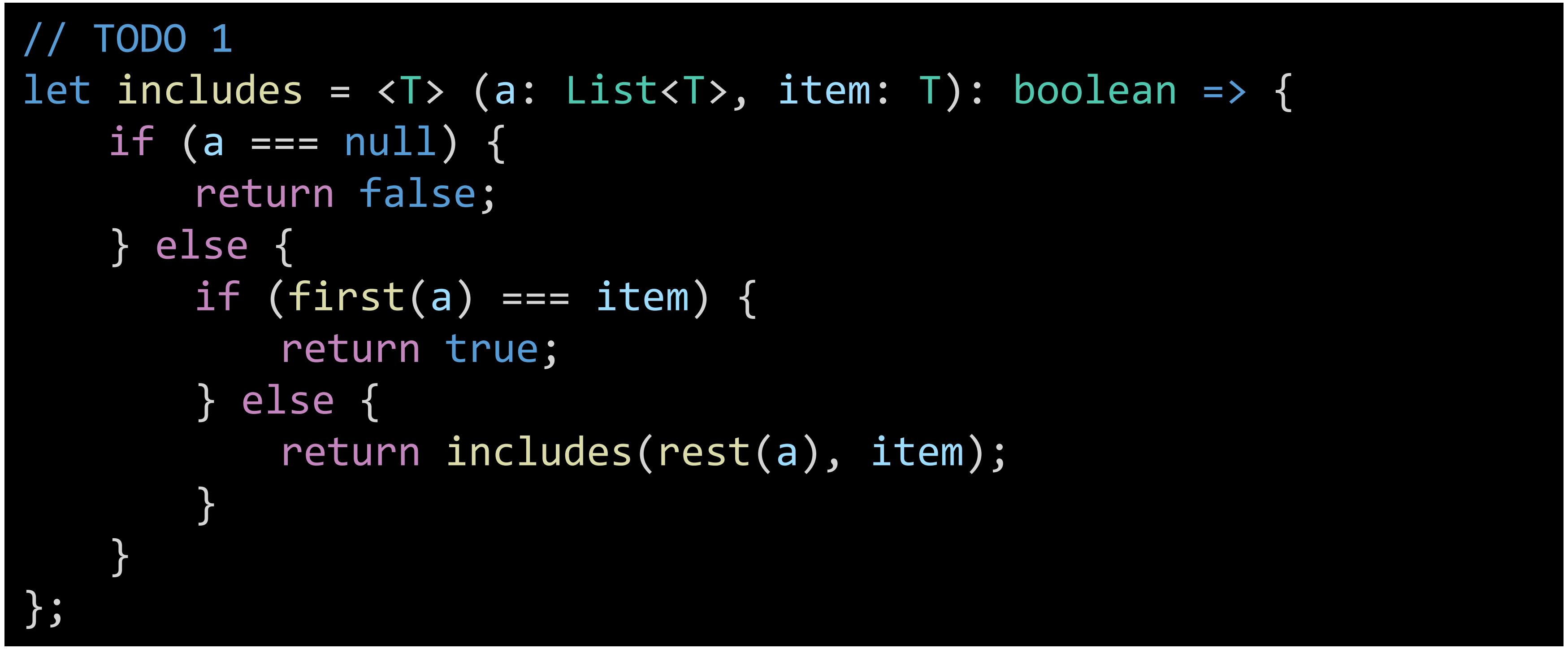
Hands-on #1: Making a Generic **includes** Function

- Open lec10 / 00-generic-functions-app.ts

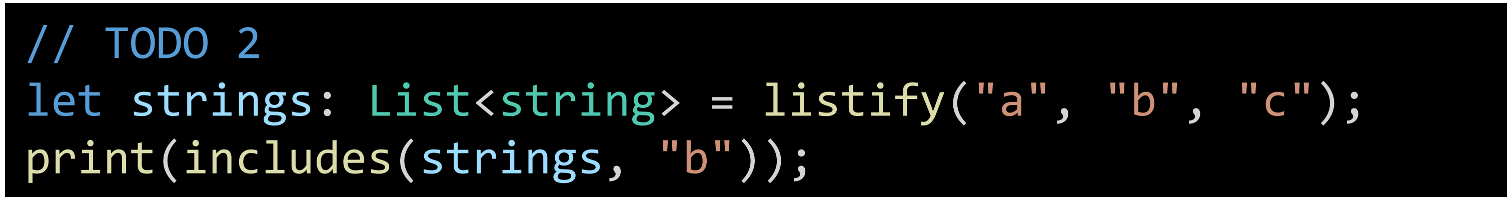
1. Convert the includes function to be a generic function.
   a) Add the diamond T syntax before the parameter list: **let includes = <T> (...**
   b) Replace the specific **number** type with the generic type **T**
      - **number** is replaced with **T**
      - **List<number>** is replaced with **List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

2. In the main function, declare a variable to hold a List of string values.
   a. Initialize it with a List of some arbitrary strings.
   b. Call the includes function using this List and a string for the 2nd argument. Print the result.

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



```
// TODO 1
let includes = <T> (a: List<T>, item: T): boolean => {
    if (a === null) {
        return false;
    } else {
        if (first(a) === item) {
            return true;
        } else {
            return includes(rest(a), item);
        }
    }
};
```



```
// TODO 2
let strings: List<string> = listify("a", "b", "c");
print(includes(strings, "b"));
```

Generic Functions (4 / 4)

- Once a function is generic, you can call the function and substitute any type for T.

```
let includes = <T> (a: List<T>, item: T): boolean => // ...
```

- Valid Function Calls:

```
includes(listify("foo", "bar"), "boz")
```

```
includes(listify(1, 2, 3), 2)
```

- For any given call, ***T can only be substituted with one type***. Invalid Function Call:

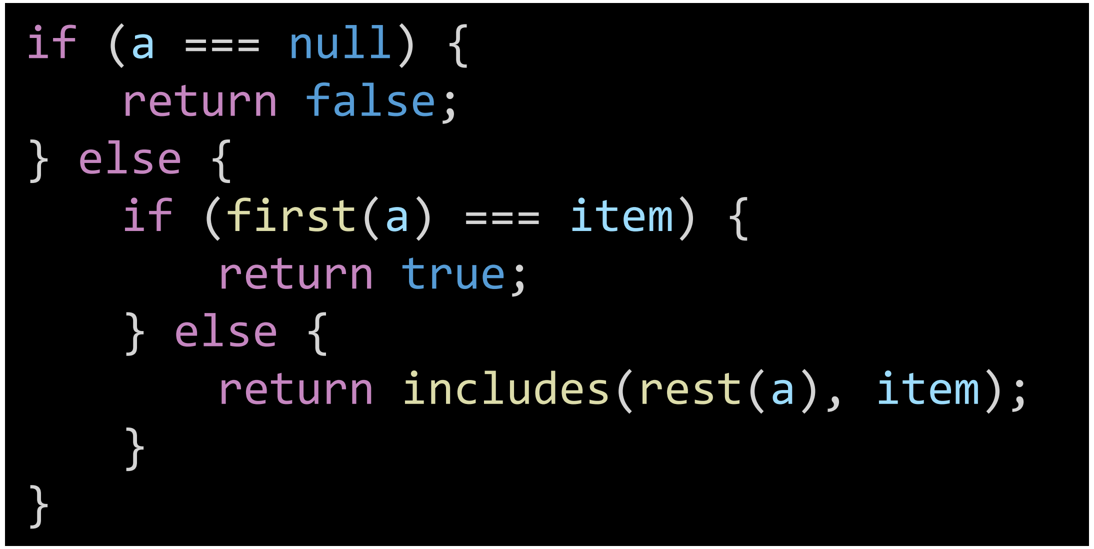
```
includes(listify(1, 2, 3), "boz")
```

- With our generic **`includes`** function, we can determine whether some value of type T is in any List holding values the same type T.

Pattern: Nesting **if-then** in an **else** Pattern

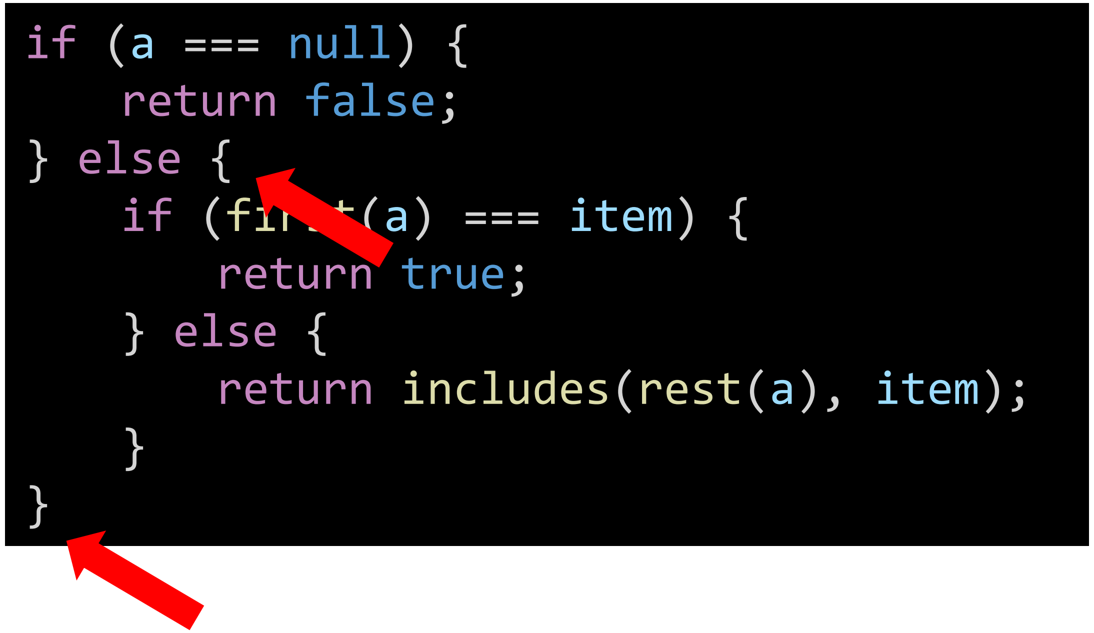
- It is commonly useful to nest additional if-then-else statements inside of subsequent else-blocks

- Why? It allows us to choose one next step from many possible options.
  - "If *this* then do X, otherwise if *that* do Y, *otherwise* do Z."

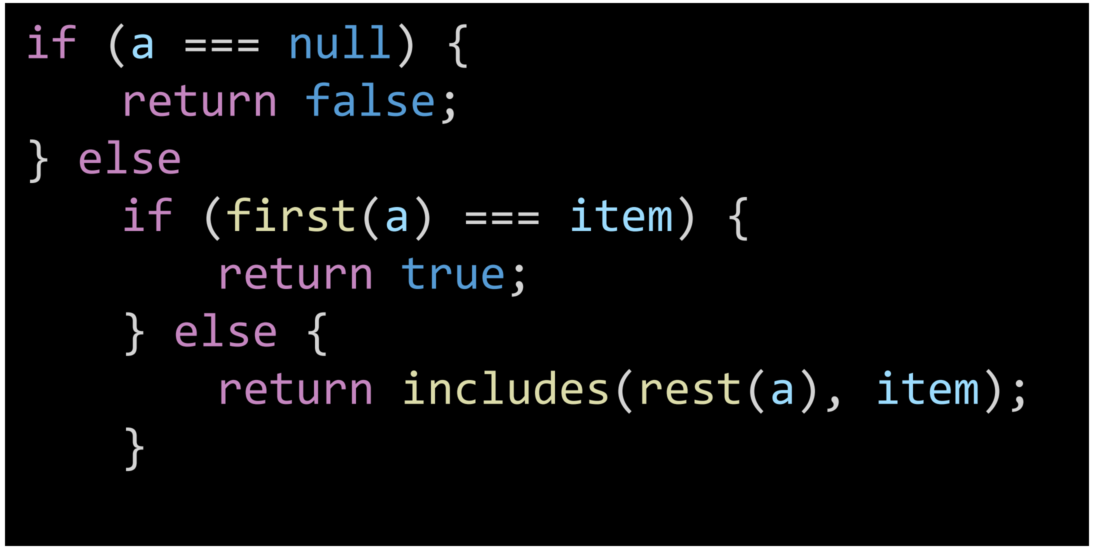


```
if (a === null) {
    return false;
} else {
    if (first(a) === item) {
        return true;
    } else {
        return includes(rest(a), item);
    }
}
```

This is so common and useful, we tend to use simpler syntax for it…



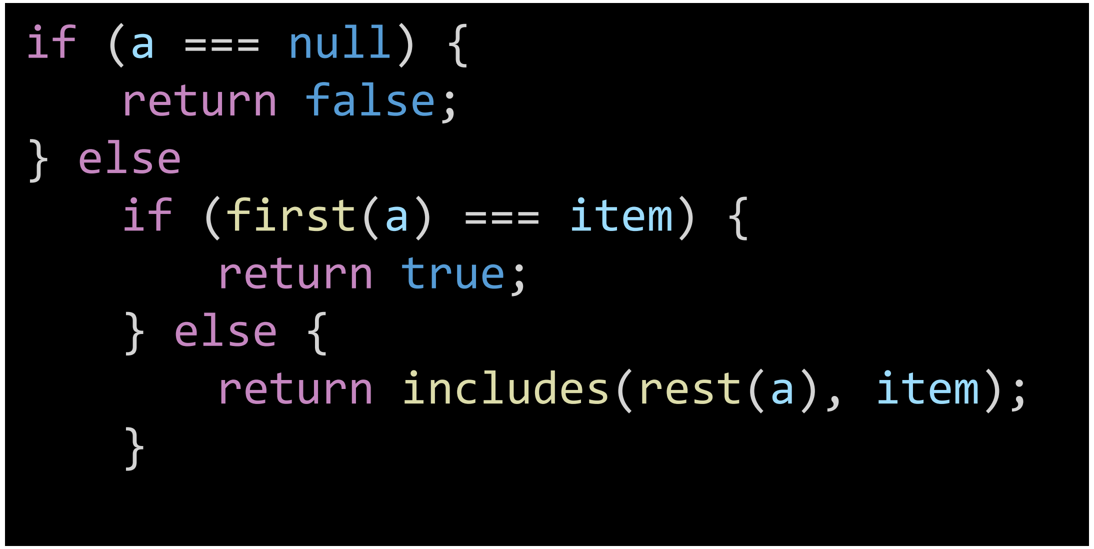
```
if (a === null) {
    return false;
} else {
    if (first(a) === item) {
        return true;
    } else {
        return includes(rest(a), item);
    }
}
```



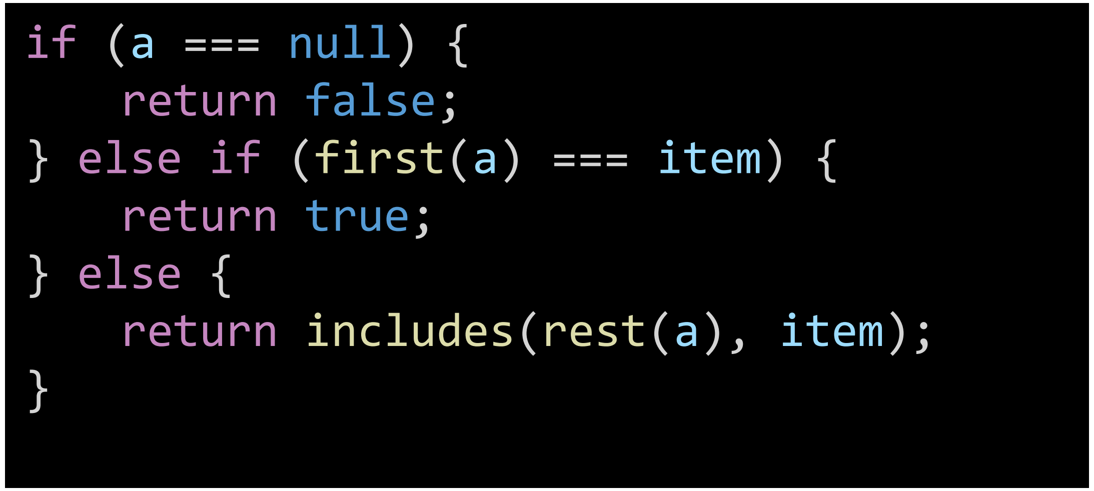
```
if (a === null) {
    return false;
} else
    if (first(a) === item) {
        return true;
    } else {
        return includes(rest(a), item);
    }
```

1. First we remove the curly braces surrounding the if-then that is nested inside of the else-block.

This is so common and useful, we tend to use simpler syntax for it…



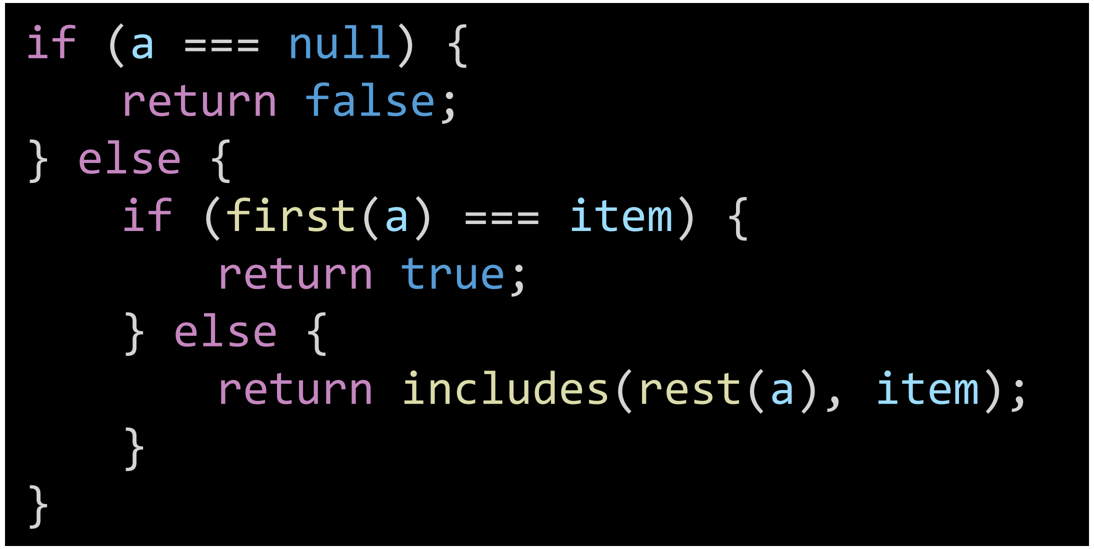
```
if (a === null) {
    return false;
} else
    if (first(a) === item) {
        return true;
    } else {
        return includes(rest(a), item);
    }
```



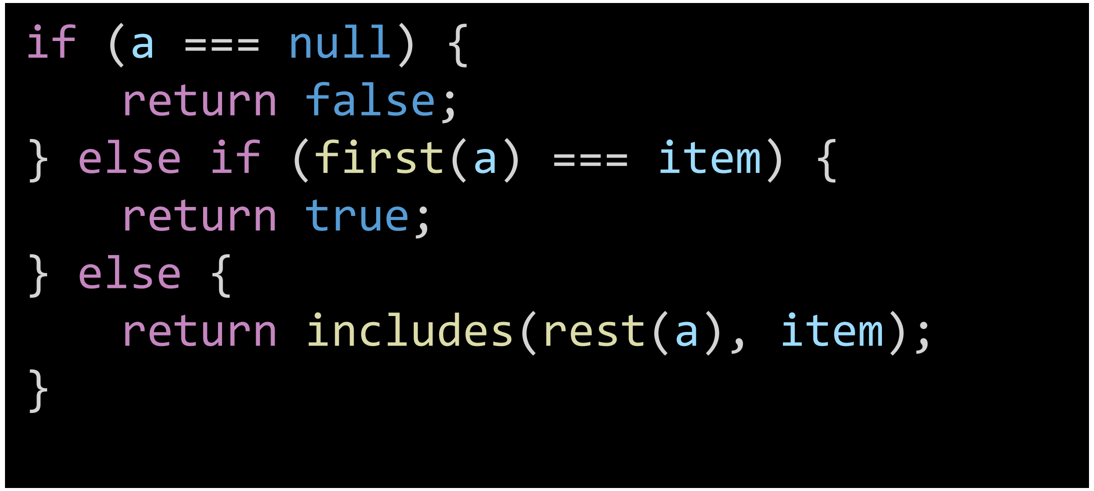
```
if (a === null) {
    return false;
} else if (first(a) === item) {
    return true;
} else {
    return includes(rest(a), item);
}
```

2. Then we clean up the spacing.

Using the **else-if** pattern is a change of *style* only. These two listings of code have the ***exact same logic***.



```
if (a === null) {
    return false;
} else {
    if (first(a) === item) {
        return true;
    } else {
        return includes(rest(a), item);
    }
}
```

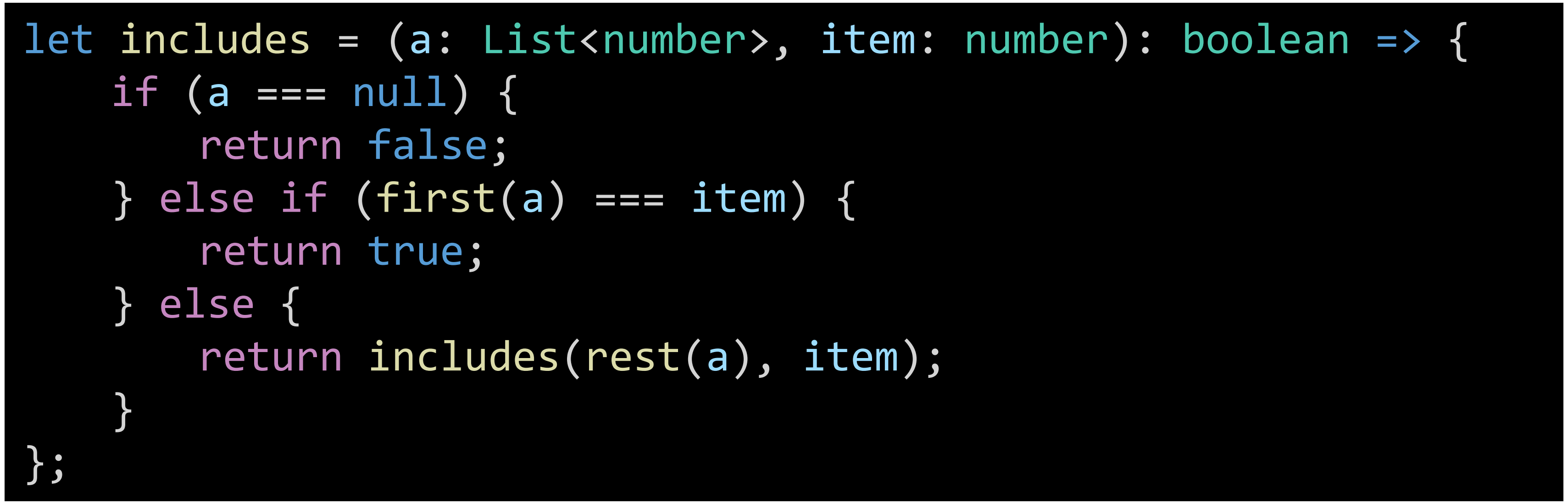


```
if (a === null) {
    return false;
} else if (first(a) === item) {
    return true;
} else {
    return includes(rest(a), item);
}
```

Notice the code is visually simpler and cleaner by using else-if.

Hands-on #2) Using the else-if Syntax Pattern

- Still in lec10 / 00-generic-functions-app.ts
- Reformat the conditional logic in the `includes` function to use the else-if syntax pattern.
- Step 1) Remove the curly brace directly following the *first* else and its matching closing curly brace.
- Step 2) Clean up the spacing by bringing the nested if to directly follow else and unindenting.
- Check-in when complete! pollev.com/compunc



```
let includes = (a: List<number>, item: number): boolean => {
    if (a === null) {
        return false;
    } else if (first(a) === item) {
        return true;
    } else {
        return includes(rest(a), item);
    }
};
```

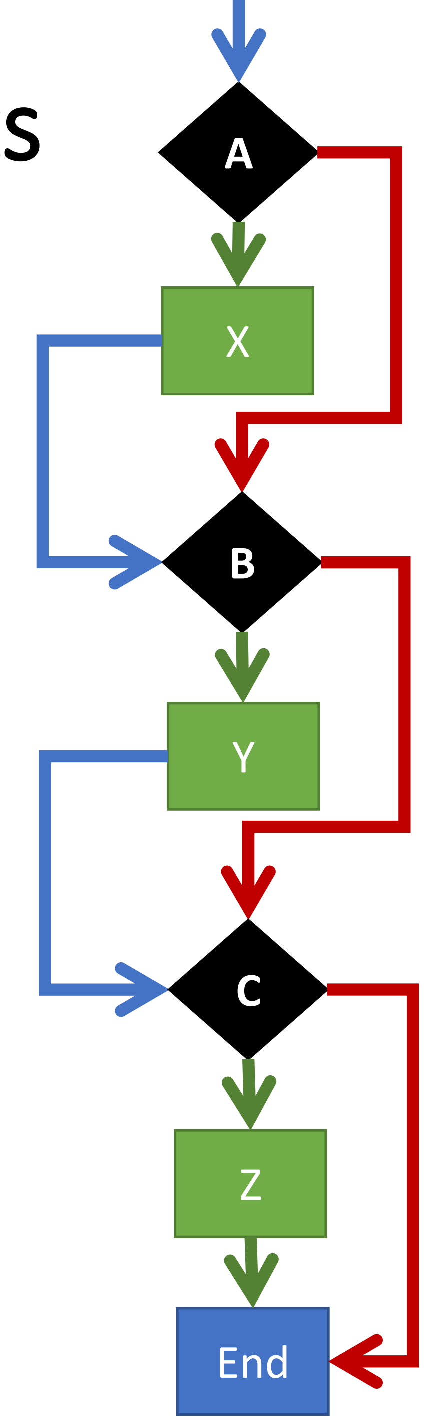
Many, independent `if-then-else` statements

- When two or more if-then-else statements are *not* nested, they are independent statements of one another.

- Each `boolean` test expression will be evaluated.

- Notice in the diagram that there is a path through *every* block X, Y, Z.

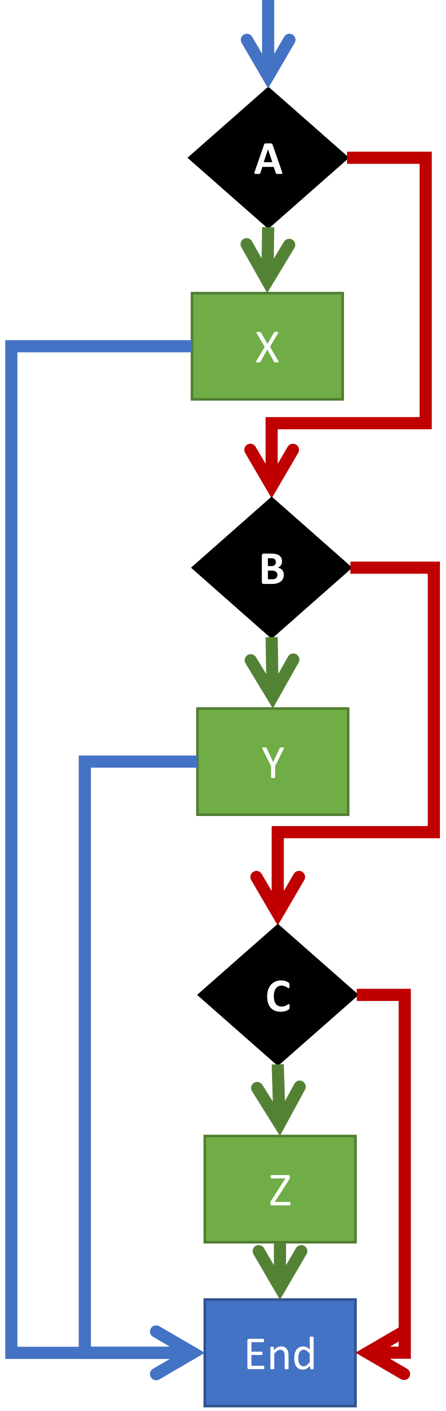
```
if (testA()) {
  print("X");
}

if (testB()) {
  print("Y");
}

if (testC()) {
  print("Z");
}

print("End");
```



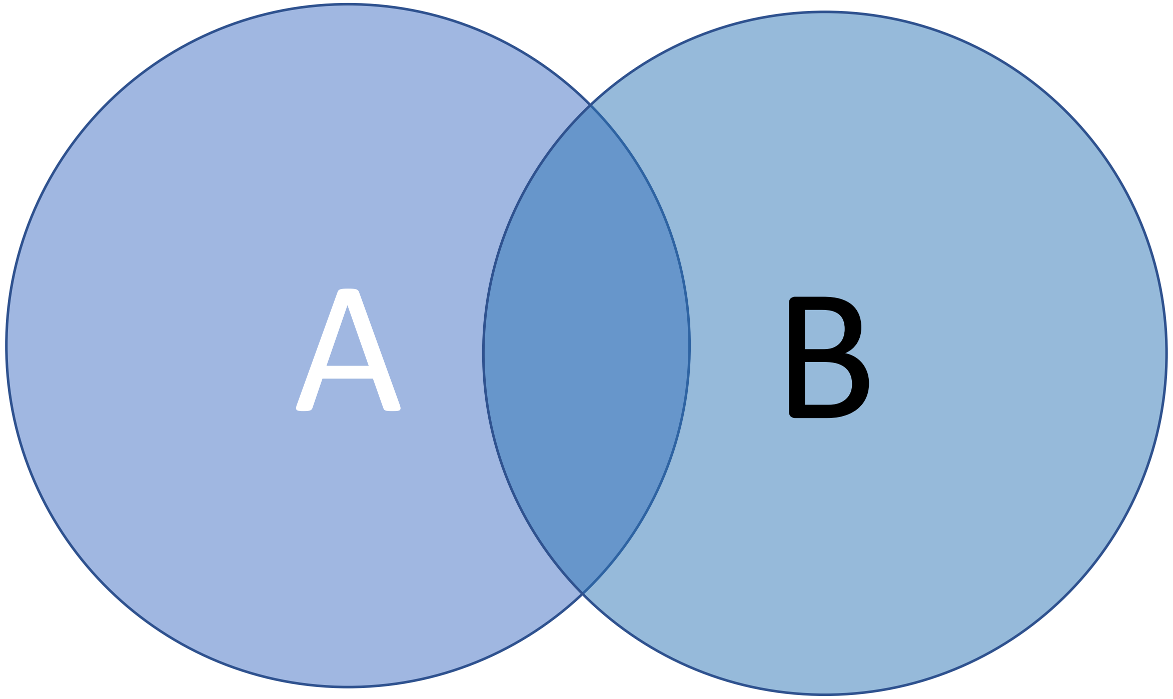
Tracing through `else-if` statements

- The previous slide does not apply to else-if statements *because…*
    - An else-if is a nested if-then
    - It is nested in the else-block
- Each `boolean` test expression will be evaluated **until one evaluates to true**. The rest are then skipped.
- Notice in the diagram that there is a path through *only one* outcome X, Y, Z.
- Useful when there are many possible next steps but you only want to choose one.

```
if (testA()) {

    print("X");

} else if(testB()) {

    print("Y");

} else if(testC()) {

    print("Z");

}

print("End");
```



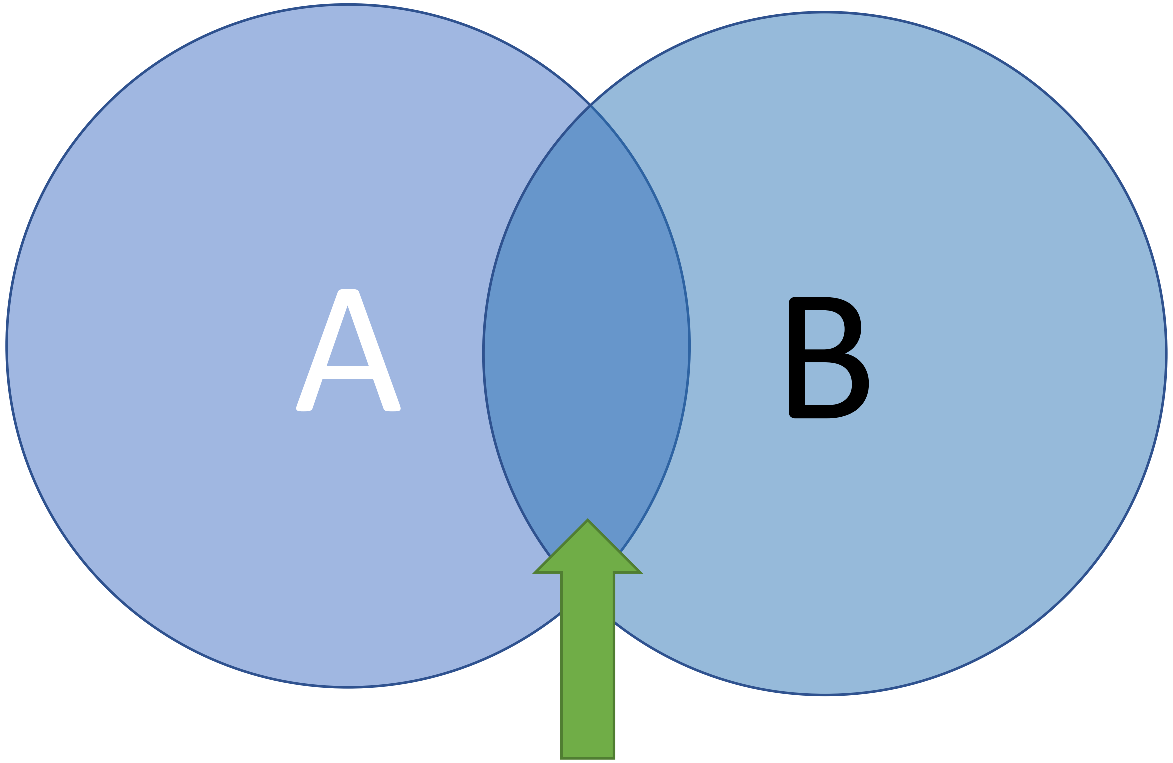
Changing Gears:
Let's Talk About Sets and Venn Diagrams



Intersection

$$\boldsymbol{A \cap B}$$

What elements are common to both A *and* B?

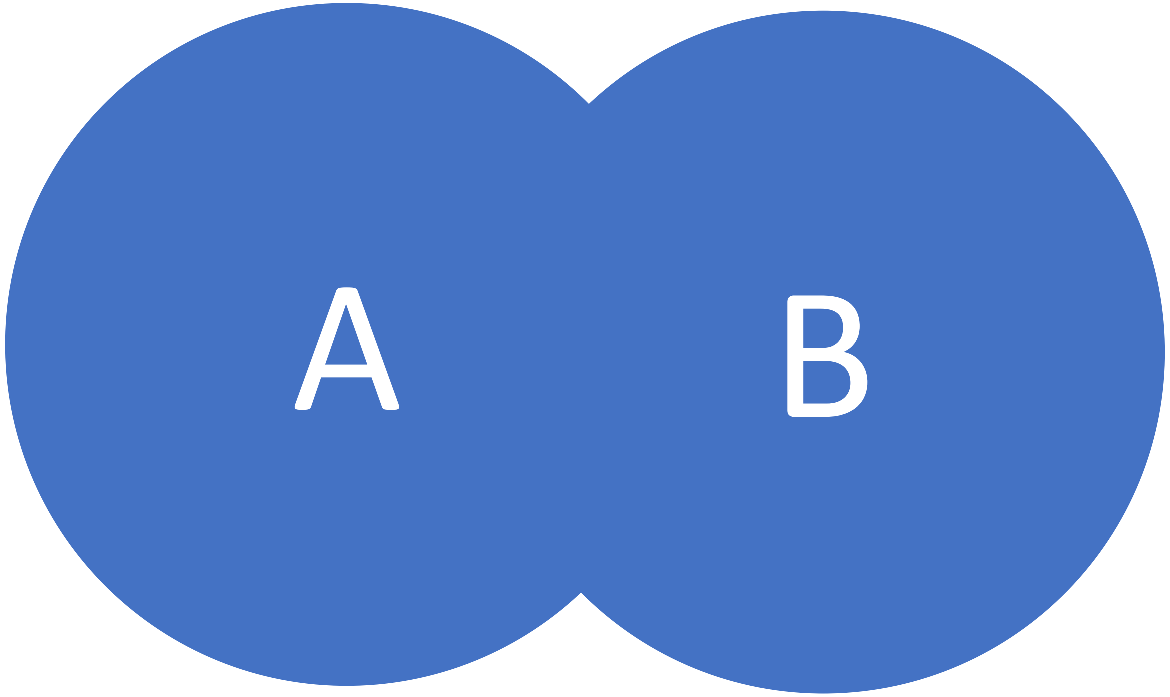


Only the *overlap* in a Venn diagram.

Union

$\boldsymbol{A} \cup \boldsymbol{B}$

What elements are either A *or* B?

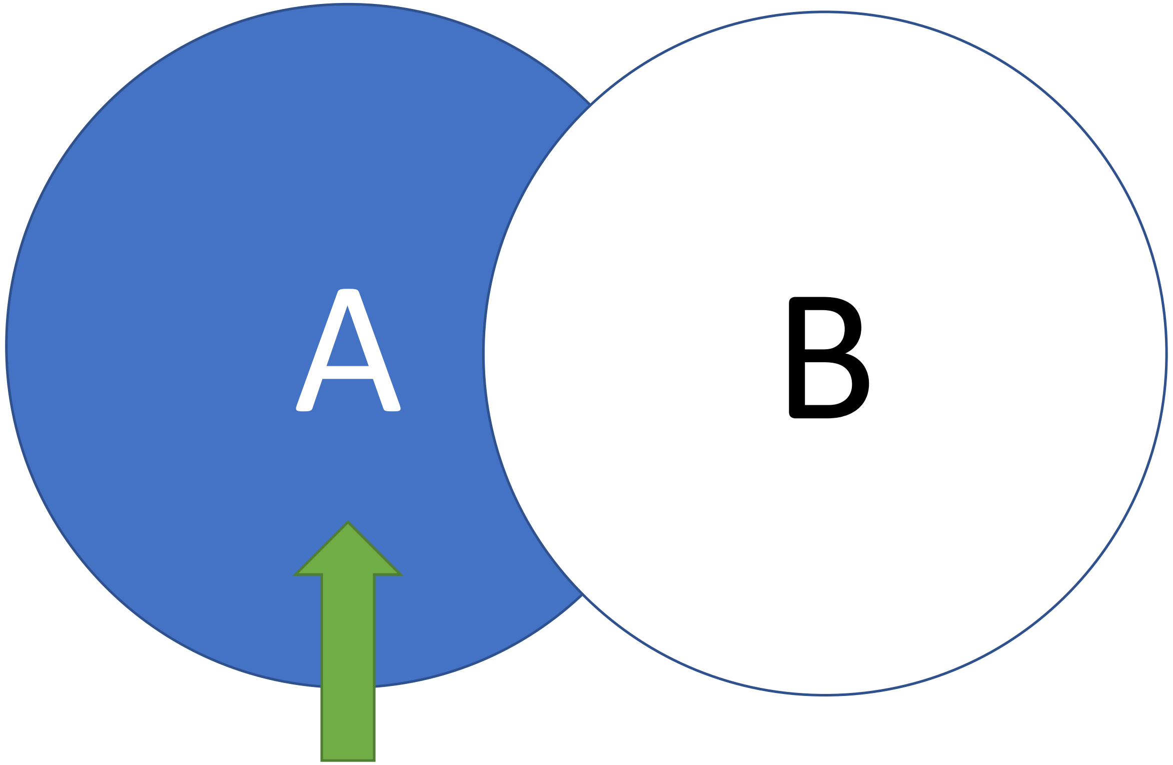


Combines the elements of both A and B.

Difference

$$\boldsymbol{A - B}$$

Remove **B**'s elements from **A.**



A without any of B's elements.

Poll Everywhere:

The set of all movies featuring Adam Sandler is $M_{Sandler}$

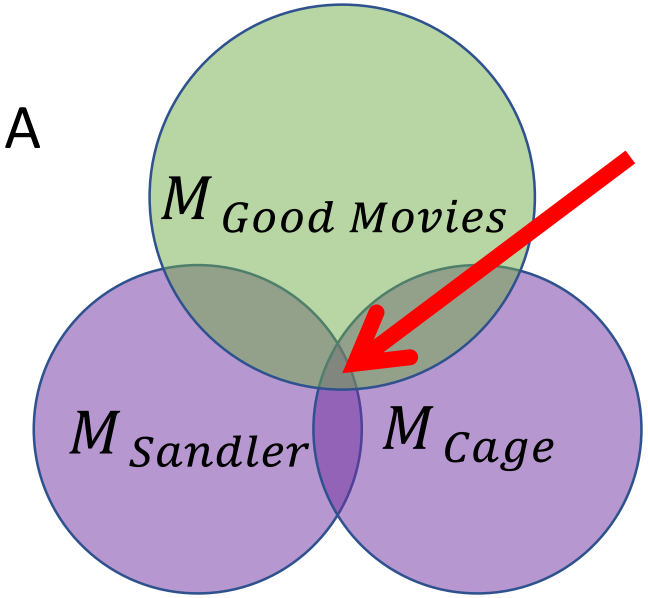
The set of all movies featuring Nick Cage is M_{Cage}

The set of all movies worth watching is $M_{Good\ Movies}$

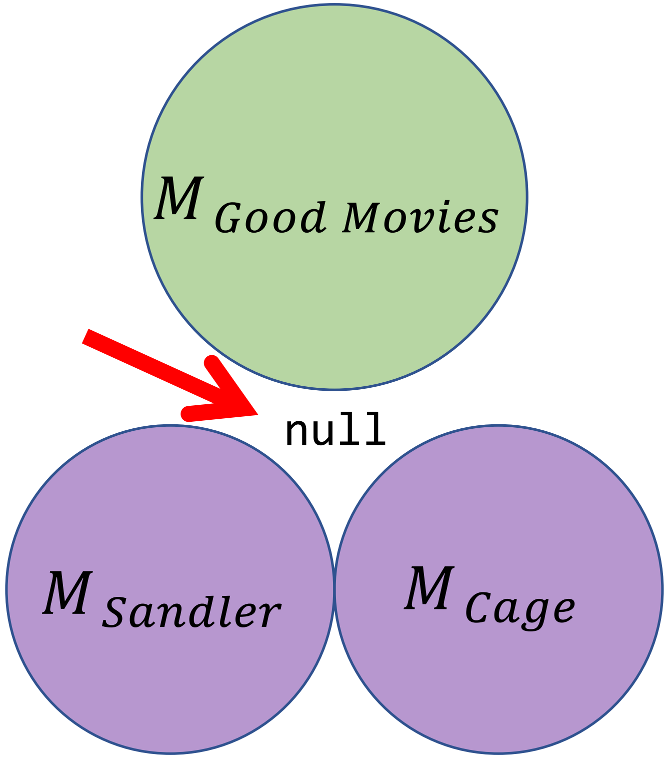
How would you represent the following set equation using a Venn diagram?

$$\left(M_{Sandler} \cup M_{Cage}\right) \cap M_{Good\ Movies}$$

A



B



Hands-on: Implementing a **`union`** Function

Open lec10 / 01-union-app.ts

Goal: Implement the *union* function.

Strategy: ***cons*** all elements of ***a*** onto a new list *ending with* ***b*** *and then prevent duplicates.*

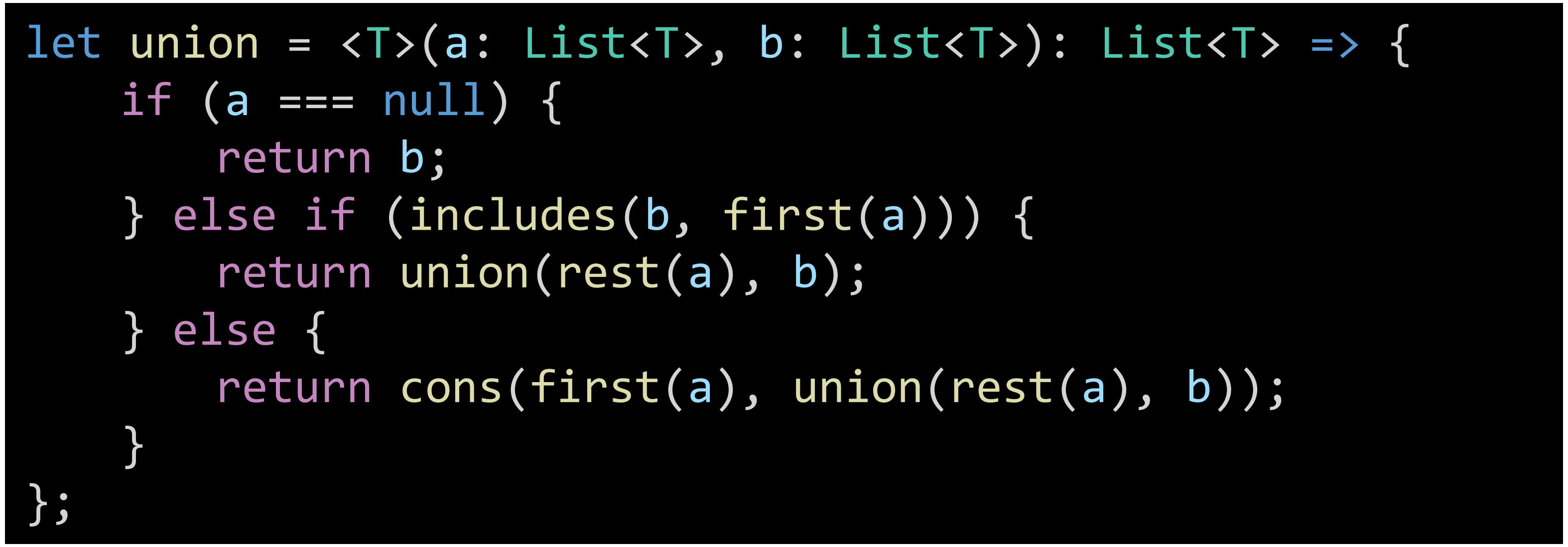
Part 1 – Combining Lists A and B

1. If **a** is null, then return **b**.
2. Else, **cons** the **first** element of **a** onto the **union** function applied recursively on the **rest of a** and **all of b**.
3. Test your program. Is it working? Check-in to say you've completed Part 1.

Part 2 – Removing Duplicates

1. Add an else-if condition after the then-block.
2. The condition of the else-if should test to see if set **b** **includes** the **first** element of **a**.
3. If it does, then recur without cons'ing. Else, do the same as Part 1, Step 2.

- Check-in to say you've completed Parts 1 and 2 once you have.



```
let union = <T>(a: List<T>, b: List<T>): List<T> => {
    if (a === null) {
        return b;
    } else if (includes(b, first(a))) {
        return union(rest(a), b);
    } else {
        return cons(first(a), union(rest(a), b));
    }
};
```

Hands-on: Using Functions to Process Data

Open lec10 / 02-movies-app.ts – use CSV file in data/movies-data.csv

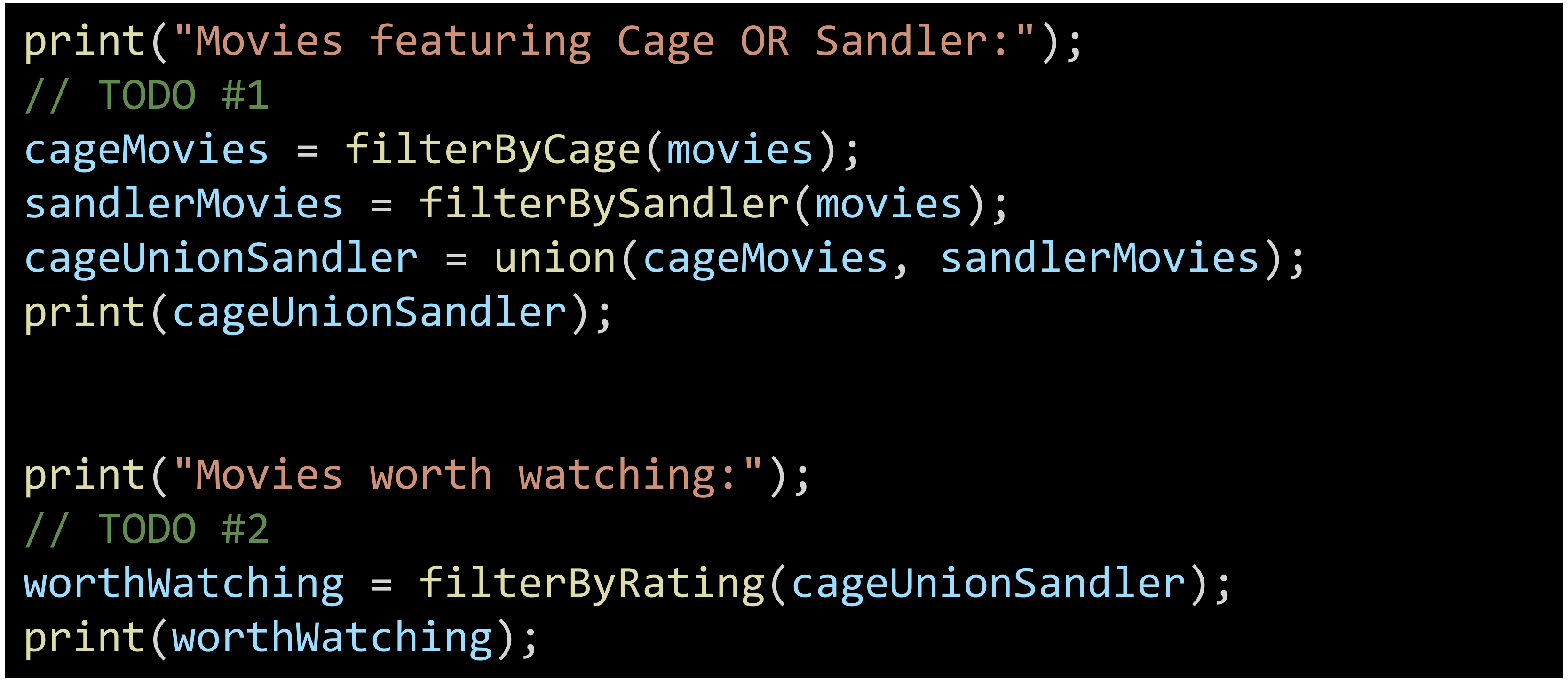
Goal: Find $\left(M_{Sandler} \cup M_{Cage}\right) \cap M_{Good\ Movies}$

1. At TODO #1
   a) assign to **cageMovies** the result of calling the **filterByCage** function
   b) assign to **sandlerMovies** the result of calling the **filterBySandler** function
   c) assign to **cageUnionSandler** the result calling the **union** function
   d) print the contents of the cageUnionSandler variable

2. At TODO #2
   a) assign to `worthWatching` the result of calling `filterByRating`… *using the List that is the union of Nick Cage's movies and Adam Sandler's movies as input.*
   b) print the contents of `worthWatching` variable

Check-in when you have the solution. pollev.com/compunc

Done? How low does `filterByRating'`s low bar need to be to find a movie?



```
print("Movies featuring Cage OR Sandler:");
// TODO #1
cageMovies = filterByCage(movies);
sandlerMovies = filterBySandler(movies);
cageUnionSandler = union(cageMovies, sandlerMovies);
print(cageUnionSandler);


print("Movies worth watching:");
// TODO #2
worthWatching = filterByRating(cageUnionSandler);
print(worthWatching);
```