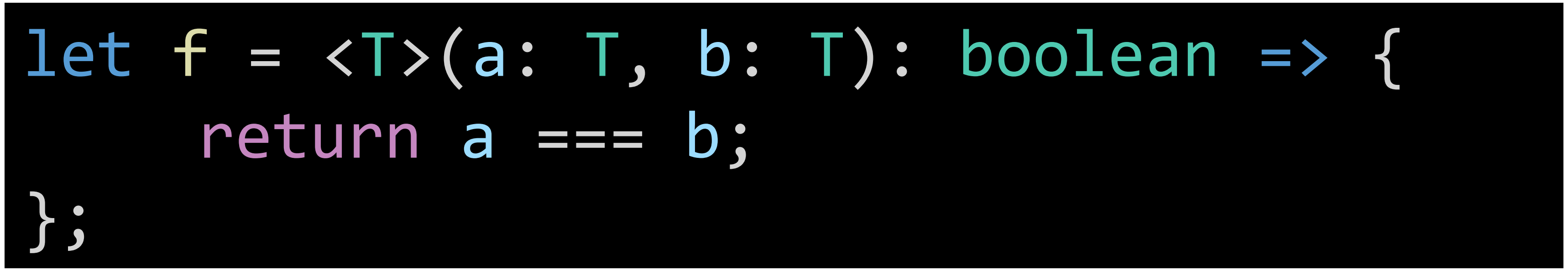
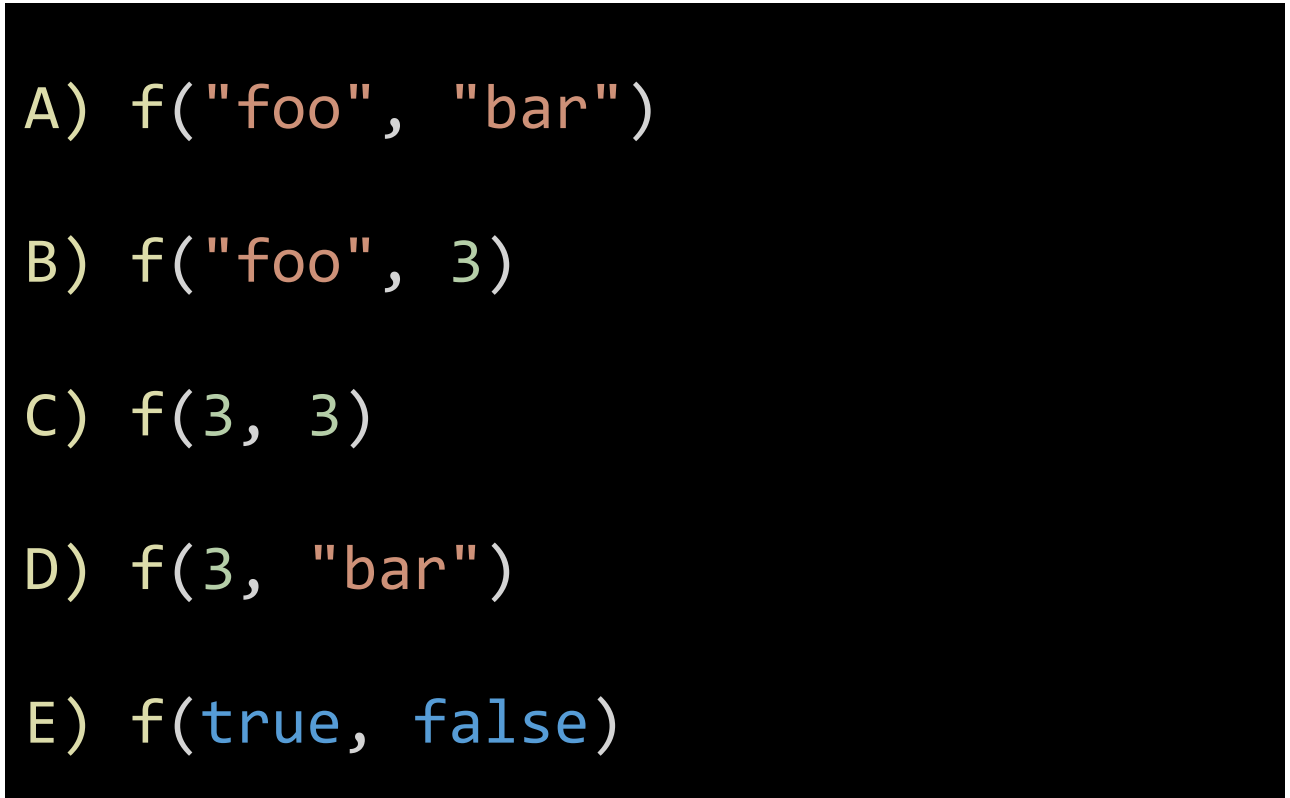
# 3. Which of the following are valid ways to call the function *f*?



```
let f = <T>(a: T, b: T): boolean => {
    return a === b;
};
```



```
A) f("foo", "bar")

B) f("foo", 3)

C) f(3, 3)

D) f(3, "bar")

E) f(true, false)
```

# Toward a Generic *filter* Function

- In the first two example files, our *filter* function worked specifically on a List of numbers *<u>or</u>* a List of string values.
  - Each used a Predicate interface that was specific to a *string* or a *number.*

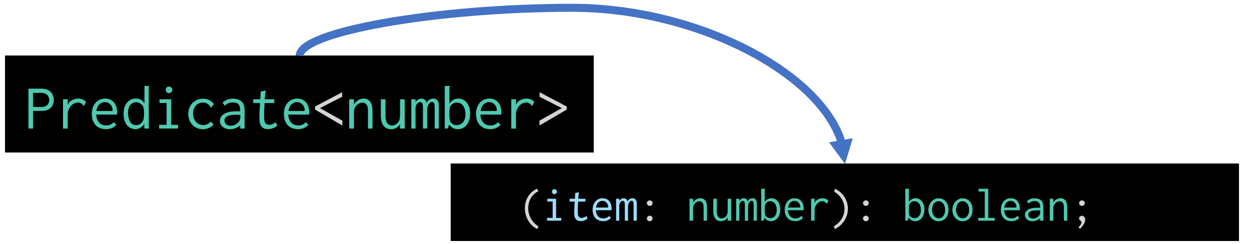
- How can we make the *filter* function generic?
  - Earlier, we introduced *generic functions* and *classes*
  - Today we'll look at generic functional interfaces
  - These ideas complement each other

# Introducing: **Generic Functional Interfaces**

- We can declare a functional interface to be generic for "any type T" by adding the diamond `<T>` after the name

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

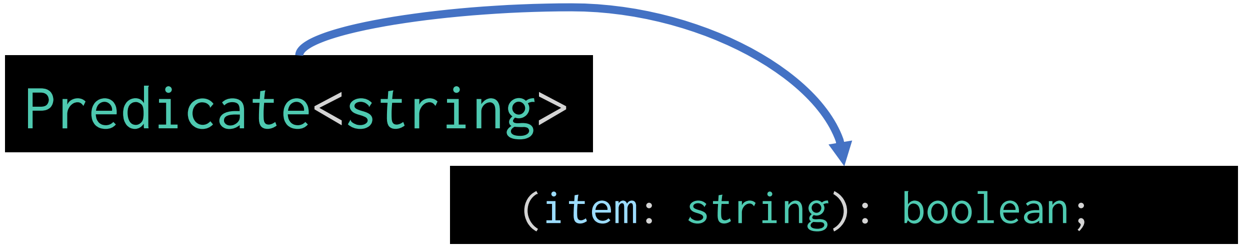
- Now, when we use type Predicate<T>, we substitute the *actual type* we want T to be.



```
Predicate<number>
```

```
  (item: number): boolean;
```

- Notice that if we have a `Predicate<string>` the function's parameter will be type `string`.



```
Predicate<string>
```

```
  (item: string): boolean;
```

# Why are **types** important?

- Types communicate *expectations* and ***capabilities*** in our programs.

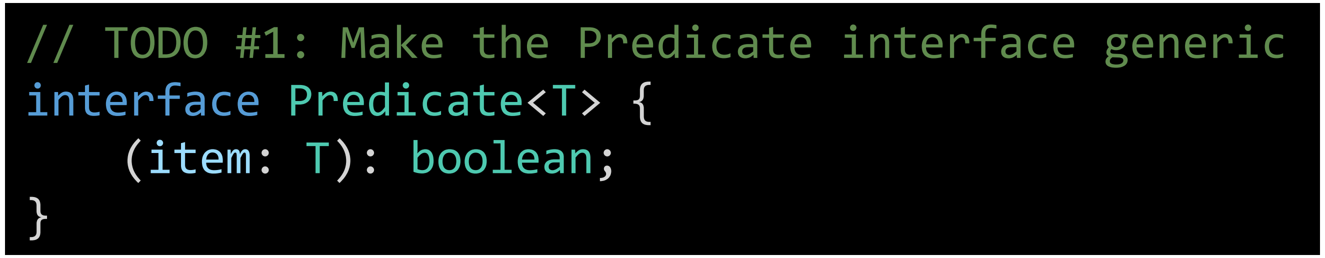
- Take the following variables, for example:

```
let item: number;
let test: Predicate<number>;
```

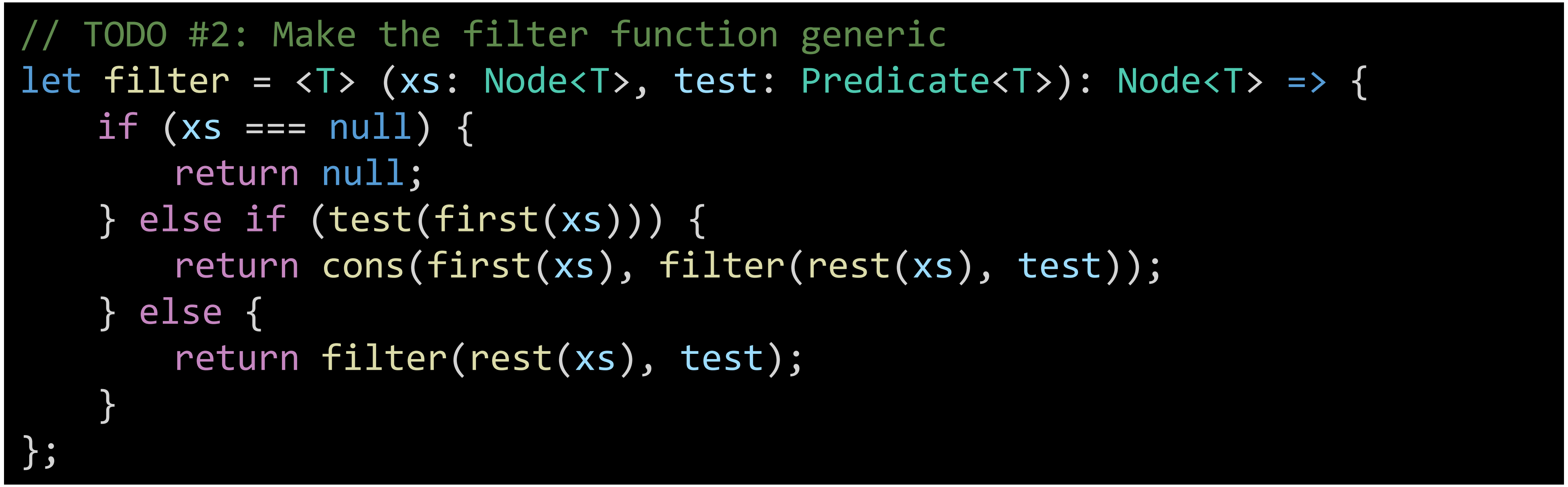
- The ways we can use ***item*** and ***test*** in our code are very different!
  - **item**: holds data whose type is number.
    With **item**, we can do the things like arithmetic, numeric comparisons, and so on.
  - **test**: holds a function that accepts a number as an input and returns a boolean. With **test**, we can **call** it as a function.

# Follow-along: Generic *Interface* & *filter*

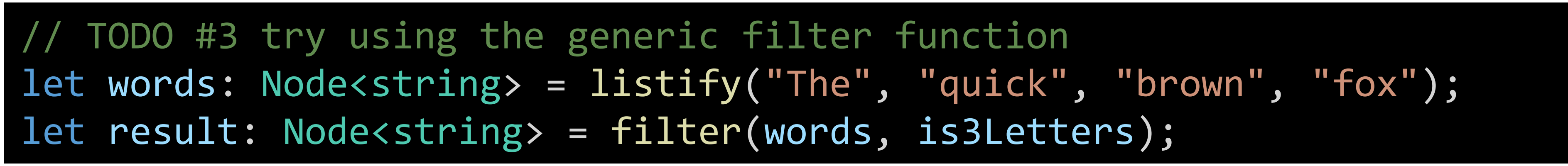
- Open 02-generic-interface-app
- TODO #1) Make the Predicate interface generic for any type T
- TODO #2) Make the filter function generic for any type T, as well
- TODO #3) Try using filter with a List of strings and a string Predicate



```
// TODO #1: Make the Predicate interface generic
interface Predicate<T> {
    (item: T): boolean;
}
```



```
// TODO #2: Make the filter function generic
let filter = <T> (xs: Node<T>, test: Predicate<T>): Node<T> => {
    if (xs === null) {
        return null;
    } else if (test(first(xs))) {
        return cons(first(xs), filter(rest(xs), test));
    } else {
        return filter(rest(xs), test);
    }
};
```



```
// TODO #3 try using the generic filter function
let words: Node<string> = listify("The", "quick", "brown", "fox");
let result: Node<string> = filter(words, is3Letters);
```

# A **Big** Idea in CS – Algorithmic Abstraction

- Once we have an algorithm, or a process for solving a problem, we can "***abstract its details away***" in a function
- If there are *values* the function needs, introduce data parameters
- If there is *logic* the function needs, introduce function parameters
  - In **filter**, the *test logic* is supplied as a function parameter
- Once we have a generic, well abstracted function… **we can reuse it!** You'll *rarely* reimplement filter logic ever again!