Functions as Parameters & Functional Interfaces

Lecture 20 – COMP110

PollEv #1) What is the difference between these two functions?

```
let filterPositives = (l: Node<number>): Node<number> => {
    if (l === null) {
        return null;
    } else if (first(l) > 0) {
        return cons(first(l), filterPositives(rest(l)));
    } else {
        return filterPositives(rest(l));
    }
};
```

```
let filterNegatives = (l: Node<number>): Node<number> => {
    if (l === null) {
        return null;
    } else if (first(l) < 0) {
        return cons(first(l), filterNegatives(rest(l)));
    } else {
        return filterNegatives(rest(l));
    }
};
```

2. Given the filter function to the left, which of the following definitions of **test** is correct?

```
let filter = (l: Node<number>): Node<number> => {
    if (l === null) {
        return null;
    } else if (test(first(l))) {
        return cons(first(l), filter(rest(l)));
    } else {
        return filter(rest(l));
    }
};
```

```
// A
let test = (e: number): string => {/*...*/}

// B
let test = (e: string): boolean => {/*..*/}

// C
let test = (e: number): boolean => {/*..*/}

// D
let test = (e: boolean): number => {/*..*/}
```

What is the difference between these two functions?

```
let filterPositives = (l: Node<number>): Node<number> => {
    if (l === null) {
        return null;
    } else if (first(l) > 0) {
        return cons(first(l), filterPositives(rest(l)));
    } else {
        return filterPositives(rest(l));
    }
};
```

```
let filterNegatives = (l: Node<number>): Node<number> => {
    if (l === null) {
        return null;
    } else if (first(l) < 0) {
        return cons(first(l), filterNegatives(rest(l)));
    } else {
        return filterNegatives(rest(l));
    }
};
```

- The same overarching logic applies to both functions: filter a List down to only items that satisfy some boolean test criteria
- The boolean test criteria is different, though!
- Wouldn't it be awesome if we could write *one* filter function and simply "pass in" this piece of logic?

Separating *filter's* algorithmic logic from its *test* criteria's (1 / 3)

- The definition of **filter** below works generally for any criteria function named **test** that can be given a number and will return a **boolean**...

```
let filter = (l: Node<number>): Node<number> => {
    if (l === null) {
        return null;
    } else if (test(first(l))) {
        return cons(first(l), filter(rest(l)));
    } else {
        return filter(rest(l));
    }
};
```

- ... but, we don't want **filter** to work for *only one* specific **test** function. We want it to work for any *kind of test function*. Two examples:

```
let isPositive = (n: number): boolean => { return n > 0; };
let isNegative = (n: number): boolean => { return n < 0; };
```

Separating *filter's* algorithmic logic from its *test* criteria's (2 / 3)

- What if we could make **`test`** a parameter of the **`filter`** function?

```
let filter = (l: Node<number>, test:  ???  ): Node<number> => {
    if (l === null) {
        return null;
    } else if (test(first(l))) {
        return cons(first(l), filter(rest(l), test));
    } else {
        return filter(rest(l), test);
    }
};
```

- Then we could define functions that are valid substitutes for **`test`**…

```
let isPositive = (n: number): boolean => { return n > 0; };
let isNegative = (n: number): boolean => { return n < 0; };
```

- … and, finally, *call* `filter` by **passing** in the `test` function to use:

```
let input: Node<number> = listify(-1, 2, -3, 0, 4);
let positives: Node<number> = filter(input, isPositive);
let negatives: Node<number> = filter(input, isNegative);
```

Separating *filter's* algorithmic logic from its *test* criteria's (3 / 3)

- What is the test parameter's type?

```
let filter = (l: Node<number>, test: ???): Node<number> => {
    if (l === null) {
        return null;
    } else if (test(first(l))) {
        return cons(first(l), filter(rest(l), test));
    } else {
        return filter(rest(l), test);
    }
};
```

- It's a *function!* But what is a function's type?

What is a function's **type**?

- Generally, any two values of the same type can be substituted for one another and the program's *type checking will remain valid*.

- Imagine you could swap the names of a two function calls...

```
let resultA: boolean = isNegative(1);
let resultB: boolean = isPositive(-1);
```

OR

```
let resultA: boolean = isPositive(1);
let resultB: boolean = isNegative(-1);
```

- If our program's *type checking* is valid after substituting function calls, then we could say that `isPositive` and `isNegative` are the same ***type of function***.

- What would need to be the same about those functions for this to be true?

A Function's **Type**

- Is made up of its parameter's types *and* its return type

(parameter$_0$,...): returnType

- For example, these two functions are the same *type* of function.

```
let isNegative = (n: number): boolean => {
    return n < 0;
}

let isPositive = (n: number): boolean => {
    return n > 0;
}
```

Introducing: **Functional Interfaces**

- A function's type is made up of its parameter type(s) *and* its return type
  - So how do we give that type a name?

```
(parameter0: type0,...): returnType
```

- A **functional interface** assigns a **Name** to a ***type of function***.

```
interface Name {
  (parameter0: type0,...): returnType;
}
```

- For example, the functional interface to the right says

  "A Predicate is any function with a single parameter of type number that returns a boolean."

```
interface Predicate {
  (element: number): boolean;
}
```

Follow-Along: Higher Order Functions

- Open lec20 / 00-predicate-app

1. Define a functional interface for type **Predicate**
   - Informally: a predicate tests whether a piece of data meets some criteria
   - It takes an argument (number, in this example) and returns a boolean

2. Add 2nd parameter named **test**, of type **Predicate**, to the filter function
   - Replace the call to isPositive to the test parameter
   - Modify the recursive calls to filter such that they pass the test parameter along

3. Modify the call to the **`filter`** function in main
   - The filter function now needs a second parameter. Try using one of the predicate functions: isPositive, isNegative, isZero.

```
// TODO #1: Define a functional interface for Predicate
interface Predicate {
    (n: number): boolean;
}
```

```
// TODO #2: Add a parameter to supply the test function
let filter = (xs: Node<number>, test: Predicate): Node<number> => {
    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 calling filter with different predicates
let output: Node<number> = filter(input, isZero);
```

Higher-order Functions:
Functions as Parameters

- **Data parameters** allow us to give a function the "extra pieces of **information** it needs"

- With **function parameters**, we can give a function the "extra pieces of **process *or* logic** it needs"
  - When you order eggs you not only ask for a specific # of eggs, but also refer to the process of cooking eggs you desire.
  - For example scrambled, over easy, sunny side up, or some custom instruction

- A function that accepts another function as a parameter is one type of a **higher-order function**

Hands-on: Implementing a `string` Predicate

- Open lec15 / 01-string-predicate-app.ts

- Notice the Predicate and filter implementations in this file work on string values

- TODO #1) Define a function named longWords
  - Parameter: a string parameter named word
  - Return Type: boolean
  - Return true when the word's length (word.length) is greater than 4

- TODO #2) Rather than filtering with the *startsWithT* predicate function, filter with your *longWords* predicate function.

- Check-in on PollEv.com/compunc when your code is working correctly.

```
// TODO #1: Define a `longWords` predicate function
let longWords = (word: string): boolean => {
   return word.length > 4;
};
```

```
// TODO #2: Change the predicate function to use longWords
let output: Node<string> = filter(input, longWords);
```

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 *or* a List of string values.
  - Each used a Predicate interface that was specific to a *string* or a *number.*

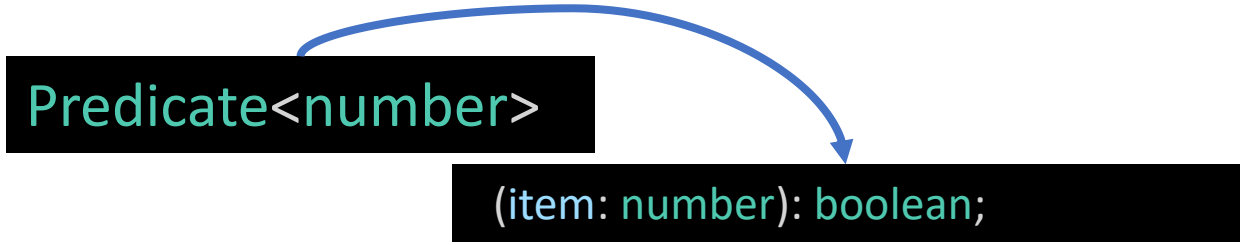
- How can we make the *filter* function generic?
  - Earlier, we introduced *generic functions* and *classes*
  - Today we'll look at generic functional interfaces
  - These ideas complement each other

Introducing: **Generic Functional Interfaces**

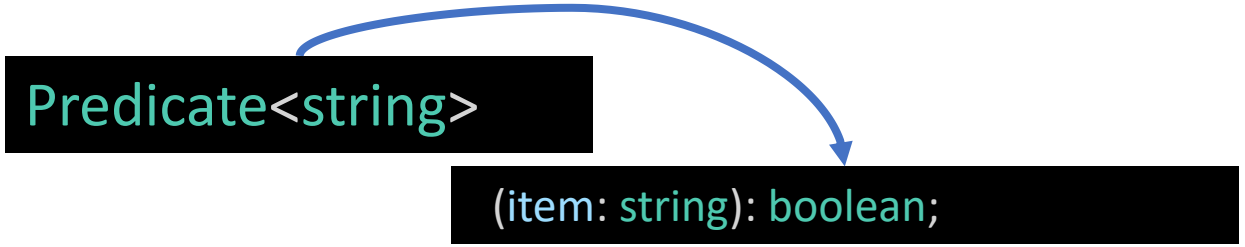
- We can declare a functional interface to be generic for "any type T" by adding the diamond `<T>` after the name

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

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



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



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!

Using **Function Literals** as Function Parameters (Preview)

Given a **`Node<string>`** value **`xs`**,

```
let xs = listify("a", "ab", "abc", "abcd");
```

and a **`Predicate<string>`** function **`isLong`**,

```
let isLong = (s: string): boolean => {
   return s.length > 3;
};
```

you can filter **`xs`** by **`isLong`**:

```
let ys = filter(xs, isLong);
```

Do you need to define a new function every time you filter data?

Using **Function Literals** as Function Parameters (Preview)

You don't *have* to declare a named `Predicate` in order to make use of `filter`.

Anywhere a function parameter is required, you can write a **function literal** instead:

```
let ys = filter(xs, (s: string): boolean => {
   return s.length > 3;
});
```

When you do not plan to use a `Predicate` again, this is *really* nice.

Other names you'll see for function literals: anonymous functions, lambdas, closures*.

* The term *closure* implies more functionality than we're revealing here, but for now you can consider them synonyms.

Type Inference with Function Parameters (Preview)

- The `filter` function is defined generically as:

```
let filter = <T> (input: Node<T>, test: Predicate<T>): Node<T> => { /* … */ };
```

- Suppose we're trying to call `filter` with a `Node<string>` input value:

```
filter(listify("hello", "world"), (item: ______): _______ => { /* … */ });
```

- Can we *infer* the exact types which must fill in the blanks?

1. We know type **T** in the **input** List must be **string** because we're giving it a **Node<string>**
2. That means the **test Predicate<T>** function must satisfy the **Predicate<string>** interface
3. Since **Predicate<T>** is defined as **(item: T): boolean**, then, it follows, the function literal's parameter and return type ***must be***:

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

Type Inference with Function Parameters (Preview)

- Higher-order functions specify their function parameters' type
  - `filter<T>` requires a `Predicate<T>` function… `(item: T): boolean`

- When a function literal is used as a parameter in a higher-order function, its parameter types and return types can be *inferred* by TypeScript.

Given input is a **`Node<string>`** values, then the function literal used below…

```
let long = filter(input, (s: string): boolean => {
    return s.length > 3;
});
```

… can be rewritten as follows and its types will be inferred by TypeScript …

```
let long = filter(input, (s) => {
    return s.length > 3;
});
```