Functions as Parameters & Functional Interfaces

Lecture 12 – COMP110 – Spring 2018

Announcements

- Next problem set will post soon and be due before Spring Break.
  - More practice with recursive functions.
- Next worksheet: not released until *after* Spring Break!
- Midterm 0 Grades
  - The average (with 99% grading complete) is…
  - Grades will post tomorrow afternoon.

PollEv #0: What is the printed output?

```
let b = (n: number): number => {
   print("b");
   return n + 1;
};

let a = (n: number): number => {
   print("a" + n);
   let result: number = b(n);
   print("a" + result);
   return result;
};

let main = async () => {
   print(a(1));
};

main();
```

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

```
let filterPositives = (l: List<number>): List<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: List<number>): List<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: List<number>): List<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: List<number>): List<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: List<number>): List<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* generic 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: List<number>): List<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 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: List<number>, test:  ???  ): List<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: List<number> = listify(-1, 2, -3, 0, 4);
let positives: List<number> = filter(input, isPositive);
let negatives: List<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: List<number>, test:   ???   ): List<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**?

- It is the type(s) and order of parameters it needs *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;
}
```

- This means you can call `isPositive` anywhere you can call `isNegative` by substituting only the name.

Introducing: **Functional Interfaces**

- A function's *type* is the combination of its parameters' types and its return type
  - So how do we give that type a name?

```
(parameter: type,...): returnType
```

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

```
interface Name {
  (parameter: type,...): returnType;
}
```

- For example, the functional interface to the right says

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

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

Follow-Along: Higher Order Functions

- Open lec12 / 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: List<number>, test: Predicate): List<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: List<number> = filter(input, isZero);
```

Higher-order Functions:
Functions as Parameters

- Previously in the semester we saw how **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 lec12 / 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 hasAnXorZ 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: List<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?
  - Last week we introduced *generic functions* (we'll review on the next slide)
  - Today we'll look at generic functional interfaces
  - These two ideas complement eachother

Generic Functions Review (1 / 2)

- **Step 1:** 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 Review (2 / 2)

- **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 => // ...
```

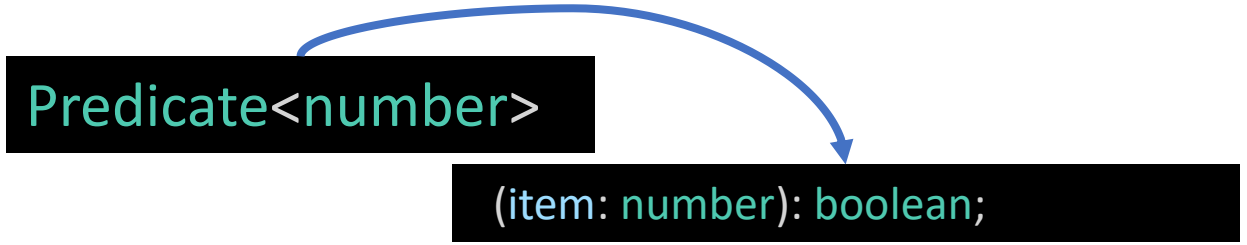
- Read as "for **any type T**, if you give the **`includes`** function any `List` of T values and an item T, it will return whether the `List` includes the `item`."

Introducing: **Generic Functional Interfaces**

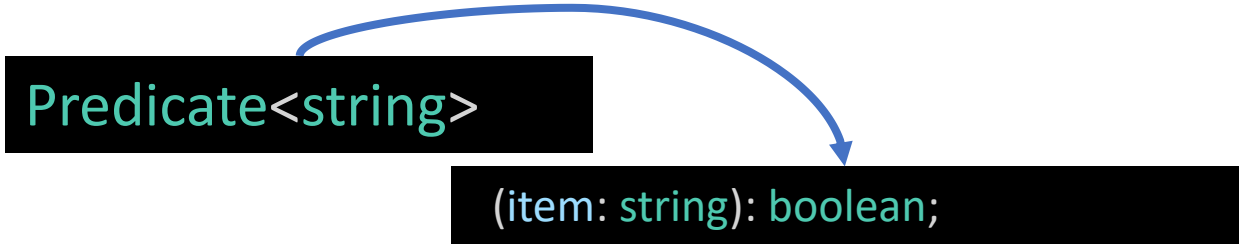
- We can declare a functional interface to be generic for "any type T" by adding the diamond `<T>` after the name

```
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 ***x*** and ***p*** in our code are very different!

  - **x**: holds data whose type is number.
    With **x**, we can do the things like arithmetic, numeric comparisons, and so on.

  - **p**: holds a function that accepts a number as an input and returns a boolean. With **p**, 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: List<T>, test: Predicate<T>): List<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: List<string> = listify("The", "quick", "brown", "fox");
let result: List<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!