Function Literals and Type Inference

Lecture 14 – Spring 2018 – COMP110

0. What is the printed output when main runs?

```
export let main = async () => {
    let xs = listify("great","day","to","be","a","tar!","heel!","daggum");
    let a = filter(xs, isLong);
    let b = map(a, initial);
    let c = reduce(b, concat, "");
    print(c);
};

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

let initial = (s: string): string => {
    return s.substr(0, 1);
};

let concat = (memo: string, s: string): string => {
    return memo + s;
};
```

Warm-up #1: Given the function **foobar**, which implements the **Transform<number, string>** interface, what is the **ys** variable's inferred type?

```
let foobar = (x: number): string => {
  return "x:" + x;
};
```

```
let xs = listify(1, 2, 3, 4, 5);
let ys = map(xs, foobar);
```

Warm-up #2: What is the functional type `filter` requires?

```
let xs = listify(1, 2, 3, 4, 5);
let ys = filter(xs, ____________________);
```

A) Predicate<string>
B) Predicate<number>
C) Transform<number, boolean>
D) Reducer<number, boolean>
E) B or C
F) B, C, or D
G) All of the Above

Using **Function Literals** as Function Parameters (1 / 2)

Given a **`List<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, map, and/or reduce data?

Using **Function Literals** as Function Parameters (2 / 2)

You don't *have* to declare a named `Predicate`, `Transform`, or `Reducer` in order to make use of `filter`, `map`, or `reduce`, respectively.

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`/`Transform`/`Reducer` 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.

Follow-along: Filtering with a **function literal**

- Open lec14 / 00-function-literal-app.ts
- Let's filter the input List of string elements to only strings of length greater than 3 using a **function literal**
- Then we'll print out the filtered List.

```
let xs = listify("great", "day", "to", "be", "a", "tar!", "heel!", "daggum");
let long = filter(xs, (s: string): boolean => {
    return s.length > 3;
});
print(long);
```

Hands-on: Mapping with a **function literal**

- Open 01-map-with-literal-function.ts

- At TODO #1, declare a variable named **initials** and assign it the result of calling the **map** function with arguments:
  1. **long** – the List of "long" input strings
  2. A **function literal** that takes in a **string as a parameter** named **s**, has a **string return type**, and returns the first character of its string parameter: s.substr(0, 1)

- Print out initials to test that your call to **`map`** is correct

- Check-in on `PollEv.com/compunc` when only the initials print

```
let initials = map(long, (s: string): string => {
       return s.substr(0, 1);
});
print(initials);
```

Warm-up #3: The `map` function requires a **`Transform<T, U>`**. Therefore, what *must* types **T** and **U** be in the example below?

```
let xs = listify("go", "tar", "heels");
let ys = map(xs, (x: _T_): _U_ => { return x.length; });
```

Type Inference with Function Parameters (1 / 3)

- The `filter` function is defined generically as:

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

- Suppose we're trying to call `filter` with a `List<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 **List<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 (2 / 3)

- Higher-order functions specify their function parameters' type
  - `filter<T>` requires a `Predicate<T>` function… `(item: T): boolean`
  - `map<T, U>` requires a `Transform<T, U>` function… `(item: T): U`
  - `reduce<T, U>` requires a `Reducer<T, U>` function… `(memo: U, item: T): U`

- 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 **`List<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;
});
```

Follow-along:
Function Parameter Type Inference

- Open 02-type-inference-app.ts

- We'll remove the types from our function parameters:

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

print("Mapped:");
let initials = map(long, (s) => {
    return s.substr(0, 1);
});
print(initials);
```

Type Inference with Function Parameters (3 / 3)

- But wait! How did we infer the U type in the last poll question (below)???

```
let input = listify("go", "tar", "heels");
let output = map(input, (x: _T_): _U_ => { return x.length; });
```

1

2

- The `map` function's declaration is:

```
let map = <T, U> (xs: List<T>, f: Transform<T, U>): List<U> => {
```

- Type T *must be* **string** because the first argument's type is `List<string>`
- How can we tell what type U must be?

1. We know it is used as the return type of our function literal
2. So it can be *inferred* using the ***`return`*** *expression* in the function's body!

Hands-on: `reduce` with a function literal

- At TODO #2 in 02-type-inference

- Declare and variable named **result** and initialize it to be the result of calling the **reduce** function with the following arguments:
  1. initials – the List of string values we mapped to
  2. a Reducer **function literal** that takes **2 parameters** named **m** and **s**, respectively, and **returns m** concatenated with **s**.
  3. an empty string "" – this is the **reduce** function's initial memo value

- Think about the reduce algorithm we covered Tuesday. Can you imagine how this reduce process works with this reducer?

- Try printing out the result variable.
- Check-in when complete

```
let result = reduce(initials, (m, s) => { return m + s; }, "");
print(result);
```

Shorthand Function Literals (1 / 2)

- Notice all of the function literal parameters are *very simple functions*

```
let long = filter(input,          (s) => { return s.length > 3; });
let initials = map(long,          (s) => { return s.substr(0, 1); });
let result = reduce(initials, (m, s) => { return m + s; }, "");
```

- What do these literals have in common?
  Each contains only a single return statement and nothing else.

- This style of function literal is so common that TypeScript provides a shorthand syntax for it which is much prettier.

Shorthand Function Literals (2 / 2)

- **IF, and only if, you are writing a function whose body contains only a single return statement**, like this function literal:

```
(s) => { return s.length > 3; }
```

- Then, you can rewrite the function using shorthand syntax. This syntactical change:
  1. Drops the curly braces
  2. Drops the return keyword
  3. Drops the semi-colon following the return statement's expression

```
(s) => s.length > 3
```

Hands-on:
Convert the literals to shorthand syntax

- In 03-shorthand-functions-app.ts

- Change each of the 3 function literals to use shorthand syntax by:
  1. Dropping their curly braces
  2. Dropping the return keyword
  3. Dropping the semi-colon following the return statement's expression

- Once complete, your program should still run the same and there should be no syntax errors…

  take a moment to admire how beautiful this is… and then check-in on pollev.

```
let long = filter(input, (s) => s.length > 3);
let initials = map(long, (s) => s.substr(0, 1));
let result = reduce(initials, (m, s) => m + s, "");
```

Follow-Along: JBII Analysis, one more time…

- Using the techniques we've learned in the past two lectures, let's try our initial example of finding an answer in the data…

- Of **games that UNC won**, how many **points** did Joel Berry **score** in total?

```
// In games that UNC won (filter)
let wins = filter(games, (g) => g.uncPoints > g.opponentPoints);

// How many *points* (map)
let points = map(games, (g) => g.points);

// Did Joel Berry Score in total? (reduce)
let total = reduce(points, (m, x) => m + x, 0);

print(total);
```

Compare these two programs…

```
export let main = async () => {
      let games = await csvToList("Select Games Data", Game);
      let wins = filter(games, (g) => g.uncPoints > g.opponentPoints);
      let points = map(games, (g) => g.points);
      let total = reduce(points, (m, x) => m + x, 0);
      print(total);

};
```

The code on the right was our initial solution to this problem when we first started exploring working with list data recursively.

Notice:

1. By *abstracting* the algorithmic concepts of *filter, map,* and *reduce* into their own functions we can *reuse* these algorithms to simplify future tasks.

2. Today's version of this program is more *declarative* in the sense it's stating the criteria and goals of what we are trying to achieve while the algorithmic details of *how* it is achieved are left to the underlying functions.

```
export let main = async () => {
      let games: List<Game> = await csvToList("Basketball Data", Game);
      games = filterByWins(games);
      let points: List<number> = gamesToPoints(games);
      let totalPoints: number = sum(points);
      print(total);
};

let filterByWins = (games: List<Game>): List<Game> => {
      if (games === null) {
            return null;
      } else {
            if (first(games).uncPoints > first(games).opponentPoints) {
                  return cons(first(games), filterByWins(rest(games)));
            } else {
                  return filterByWins(rest(games));
            }
      }
};

let gamesToPoints = (games: List<Game>): List<number> => {
      if (games === null) {
            return null;
      } else {
            let game: Game = first(games);
            return cons(game.points, gamesToPoints(rest(games)));
      }
};

let sum = (values: List<number>): number => {
      if (values === null) {
            return 0;
      } else {
            return first(values) + sum(rest(values));
      }
};
```