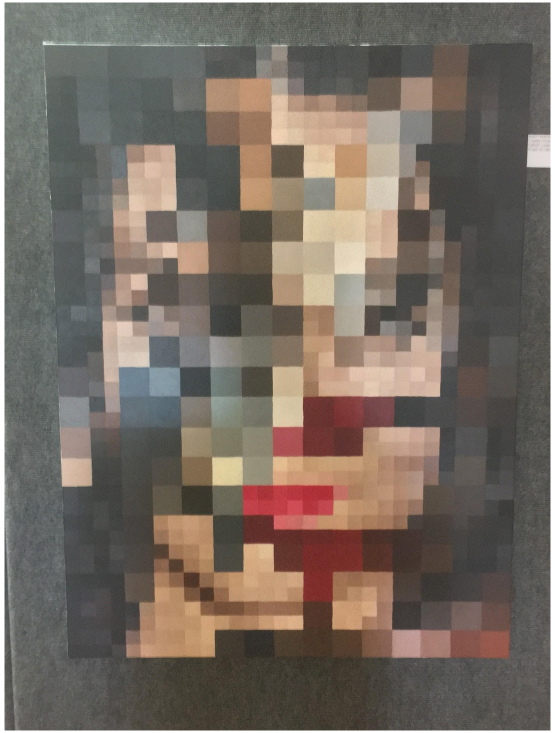
Review Session 6

FUNctional Interfaces!

About Me:

- Sophomore
- Studio Art BA, Computer Science BA
- From: Wilmington, NC
- Hobbies: biking, drawing, drinking coffee



Outline

- Generic Functions
- Functional Interfaces
- Generic Interfaces
- Filter Map Reduce

Generic Functions

- Generic functions are functions that incorporate elements of some generic type rather than a specific type
- In place of a specific type we use the symbol “T”
- Generic functions allow us to avoid duplicating logic. (Never write the same thing twice!)
- Rule of Thumb: When 2 or more functions differ only in parameter types or return type, you can replace them with a single generic function.
- Let’s consider a motivating example:

Let’s Talk About Books and Movies

- Generic functions allows us to process inputs with different types, without having to write out a bunch of superfluous code!

```
class Movie {
  title: String;
  runTime: number;
}


class Book {
   title: String;
   pages: number;
}
```

The “includes” function

- Say that I work in a store that sells both books and movies, and I need to keep track of my inventory. If I have a list of books and a list of movies, I want to see if each list contains a specific book or a specific movie.
- For this, we can use the “includes” function, which takes in a list or books or movies and an individual book or movie, and returns a boolean.
- True if the book or movie can be found in our list, false otherwise.

Generic Functions: Includes

```
let includesBook = (list: List<Book>, book: Book): boolean => {
   if (list === null) {
       return false;
   } else if (first(list) === book) {
       return true;
   } else {
       return includesBook(rest(list), book);
   }
};
let includesMovie = (list: List<Movie>, movie: Movie): boolean =>
{
   if (list === null) {
       return false;
   } else if (first(list) === movie) {
       return true;
   } else {
       return includesMovie(rest(list), movie);
   }
};
```

As you can see, there is a lot of repetition of logic! We can save ourselves some time by using Generics!

Generic Functions

One function for the price of two!

Syntax

- To specify a function is generic, we must include a set of diamond brackets containing T right before the parenthesis containing the parameters of the functions
  - <T>(parameters).....
- We use T anywhere we want to use the generic type instead of a specific type
- We could use other symbols, but conventionally we will use T. Ex:

```
let name = <T>(parameter1:List<T>, parameter2: T): returnType =>{
   //
};


let name = <T>(parameter1:List<T>): returnType =>{
   //
};
```

Example of the generic "includes" function

```
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);
   }
};
```

The Generic Includes Function

```
let movies: List<Movie> = listify(crusoe, potter, legally, boys,
panther);

let books: List<Book> = listify(perks, diary, shades,
encyclopedia, betrayal)

print(includes(movies, boys));
print(includes(movies, panther));
```

Now one function can do the same work as two!

Generic Functions: Practice

- Write a function that counts the number of occurrences of an element in a list of elements
  - Input: a list of type T and an element of type T
  - Output: a number representing the number of times that specific element occurred in the List
  - Hint: think about how we implemented includes

Solution:

```
let count = <T>(a: List<T>, item: T): number => {
   if (a === null) {
       return 0;
   } else  if (first(a) === item) {
       return 1 + count(rest(a), item);
   } else {
       return count(rest(a), item);
   }
};
```

Functional Interfaces

- Every function has a type defined by its parameters and return type
  - These two functions are *the same type of function* because they both take in a number and return a number. (The type is in purple)

```
let always7 = (n : number) : number => {
   return 7;
};

let square = (n : number) : number => {
   return n*n;
};
```

- Functional Interfaces allow us to give function types a name
- Functional Interfaces allow us to take in functions as parameters

Functional Interfaces

- Functional Interface parts:
  - Keyword **interface**
  - **Name** that is generally capitalized
  - The parameters and the return type together indicate the type of function.
  - A set of brackets containing:
    - A set of parentheses containing a comma separated list of **parameters**
    - A **return type**

Example:

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

- Example

```
interface Math {
      (x:number, y:number): number;
}

//both add and multiply are functions of type Math

let add= (num1: number, num2:number):number =>{
      return num1+num2;
};

let multiply= (a:number, b:number):number =>{
      Return a*b;
};
```

Functional Interfaces: Practice

- Part 1: Books
  - Write a functional interface that
    - Is called BookPredicate
    - takes in a Book
    - returns a boolean
  - Write a function called “ lessThan200” that returns true if the book has less than 200 pages, and false otherwise.

  Part 2: Movies

  - Write a functional interface that
    - Is called MoviePredicate
    - Takes in a Movie
    - Returns a boolean
  - Write a function called “startsWithB” that returns true if the movie title starts with B and false otherwise. You should use the function string.startsWith() with “B” as your. argument

Solution:

```
interface BookPredicate {
    (item: Book): boolean;
}

interface MoviePredicate {
    (item: Movie): boolean;
}
```

```
let lessThan200 = (b: Book): boolean => {
    if(b.pages < 200){
        return true;
    }else{
        return false;
    }
};
```

```
let startsWithB = (b: Movie): boolean => {
    if(b.title.startsWith("B")){
        return true;
    }else{
        return false;
    }
};
```

This Seems Redundant...

- Programmers hate typing the same code over and over again.
- The functional interfaces that we just made are *very* similar to each other… wouldn't it be great if we could consolidate them into one interface?
- We can! Therein lies the beauty of generics.

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

Attendance

https://tinyurl.com/110review6

Generic Functional Interfaces

- We can also use Generics in functional interfaces
- This allows us to generalize types of functions by pattern
- Generic Functional Interface parts:
    - Keyword **interface**
    - **Name** that is generally capitalized
    - A set of diamond brackets containing a comma separated list of generic types
        - <T>
        - <T, U>
    - A set of brackets containing:
        - A set of parentheses containing a comma separated list of **parameters** which could be specified types or generic types
        - A **return type** which could be a specific type or a generic type
- Example:

```
interface Name<T, U>{
        (parameter: T, parameter: U) :  U;
}
```

Generic Functional Interfaces: Examples

```
interface Example1<T,U>{
   (x:T, y:U): U;
}

let fun1=(foo:string, bar:number):number =>{
//
};

let fun2=(foo:number, bar:string):number=>{
//
};

let fun3=(foo:string, bar:string):string=>{
//
};
```

- Which of these functions is of type Example1?
  - fun1?
    - Yes! The function fun1 matches the pattern!
  - fun2?
    - Nope! If we line up the T's and U's, we see that the second parameter needs to match the return.
  - fun3?
    - Yep! The T and U don't necessarily have to be different types.

Generic Filter-Map-Reduce

- In the past we have written several filter and mapping functions that differ by testing criteria and by type
- We can use generics and functional interfaces to avoid duplication of logic
- Reminder:
  - **Filter** functions take in a list and return a list filtered by some criteria
  - **Map** functions take in a list of values and return a corresponding list of values that are transformed in some way
  - **Reduce** functions take in a list of values and return a single value

Filter and Predicate

- The generic filter function takes in two parameters:
  - A **list of type T**
  - A **function of type Predicate<T>**
- Filter returns a **filtered list of type T**
- Predicate<T> a generic functional interface that describes functions that:
  - Takes in an **element of type T**
  - Return a **boolean**
- When you call filter, you need to make sure that the **T of the Predicate** function matches the **T of the list argument**
  - You will be calling the predicate function on each element of that list, so it makes sense that the list's type should match the parameter type of the Predicate function

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


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);
    }
};
```

Walk through of Filter

```
let crusoe: Movie = new Movie();
crusoe.title = "Robinson Crusoe";
crusoe.runTime = 90;

let potter: Movie = new Movie();
potter.title = "Harry Potter and the Sorcerer's Stone";
potter.runTime = 152;

let boys: Movie = new Movie();
boys.title = "Bad Boys II"
boys.runTime = 157;

let legally: Movie = new Movie();
legally.title = "Legally Blonde";
legally.runTime = 96;

let panther: Movie = new Movie();
panther.title = "Black Panther";
panther.runTime = 134;
```

```
let filter = <T> (a: List<T>, p: Predicate<T>): List<T> => {
   if (a === null){
       return null;
   }else if(p(first(a))){
       return cons(first(a), filter(rest(a), p));
   }else{
       return filter(rest(a), p);
   }
};

let movies: List<Movie> = listify(crusoe, potter, legally, boys, panther);


print(filter(movies, startsWithB));
```

What is printed here?

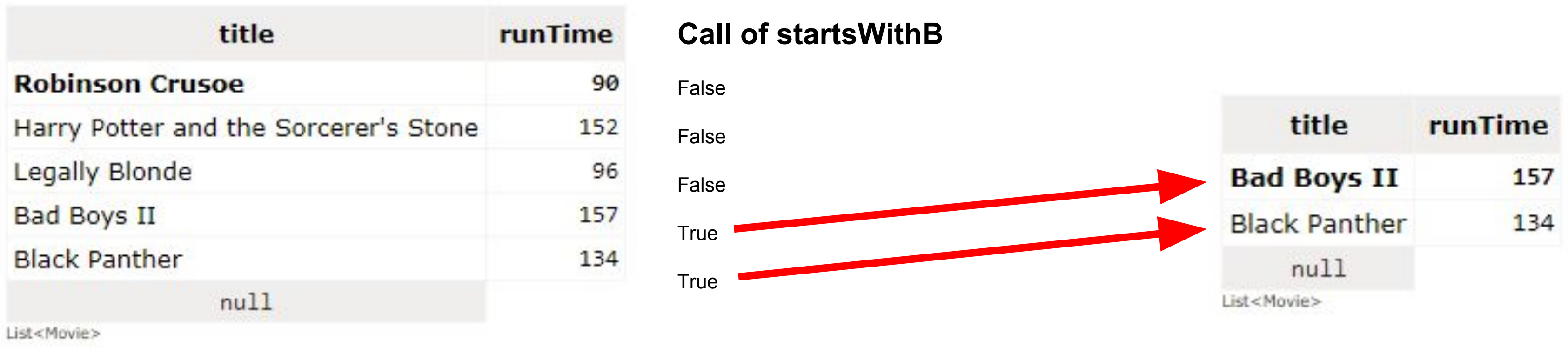
Filtering a List

Original List of Movies:

“Robinson Crusoe” -> “Harry Potter” -> “Bad Boys II” -> “Legally Blonde” -> “Black Panther”

Filtered List of Movies:

“Bad Boys II” -> “Black Panther”



title	runTime
Robinson Crusoe	90
Harry Potter and the Sorcerer's Stone	152
Legally Blonde	96
Bad Boys II	157
Black Panther	134
null	

List<Movie>

Call of startsWithB

False

False

False

True

True

title	runTime
Bad Boys II	157
Black Panther	134
null	

List<Movie>

Map and Transform

- The generic map function takes in two parameters:
  - A **list of type T**
  - A **function of type Transform<T,U>**
- Map functions return a **list of type U**
- Transform<T,U> is a generic functional interface that describes functions that:
  - Take in **an element of type T**
  - Return **an element of type U**
- When you call map, you need to make sure that the T of the Transform function matches the T of the list argument
  - You will be calling the transform function with on each element of that list, so it makes sense that the list's type should match the parameter type of the Transform function

```
interface Transform<T, U> {

   (item: T): U;

}

let map = <T, U>(xs: List<T>, f: Transform<T, U>): List<U> => {
   if (xs === null) {
       return null;
   } else {
       return cons(f(first(xs)), map(rest(xs), f));
   }
};
```

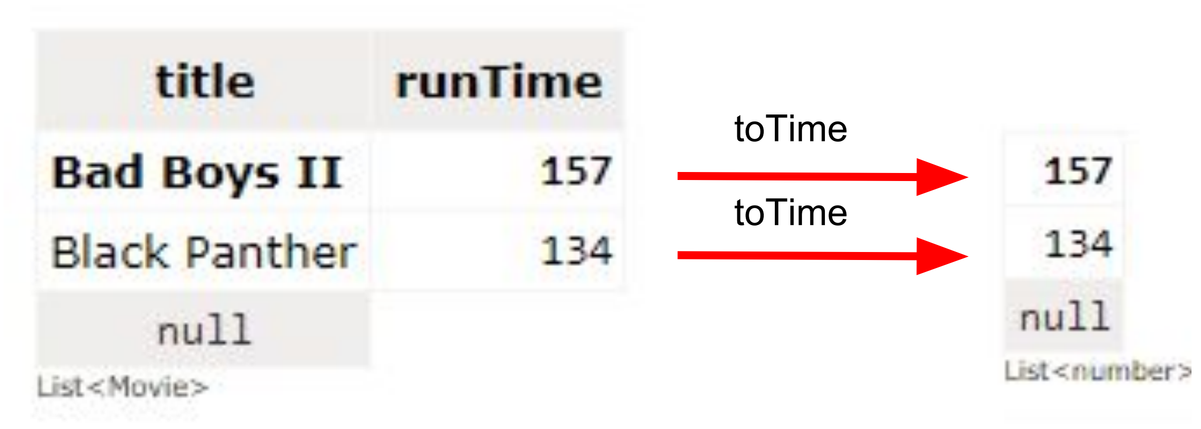
Walkthrough of Map

```
interface Transform<T, U> {
   (item: T): U;
}

let map = <T, U>(xs: List<T>, f: Transform<T, U>): List<U> => {
   if (xs === null) {
       return null;
   } else {
       return cons(f(first(xs)), map(rest(xs), f));
   }
};


let toTime = (movie: Movie): number => {
   return movie.runTime;
};
```

```
let movies: List<Movie> = listify(crusoe, potter, legally,
boys, panther);

let bMovies: List<Movie> = filter(movies, startsWithB);

let runTimes: List<number> = map(bMovies, toTime);

print(bMovies);

print(runTimes);
```

What is printed here?

map(bMovies, toTime)



title	runTime
Bad Boys II	157
Black Panther	134
null	

List<Movie>

157
134
null

List<number>

bMovies

runTimes

Reduce and Reducer

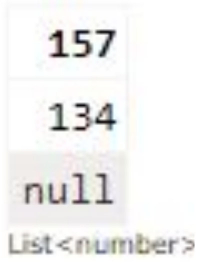
- The generic reduce function takes in three parameters:
  - A **list of type T**
  - A **function of type Reducer<T,U>**
  - A **memo of type U**
- Reduce functions return a single element of type U
- Reducer<T,U> is a generic functional interface that describes functions that:
  - Take in an **element of type U and an element of type T**
  - Return an **element of type U**
- When you call reduce, you need to make sure that the T and U of the list and memo match the T and U of the Reducer function

```
interface Reducer<T, U> {
   (memo: U, item: T): U;
}

let reduce = <T, U>(xs: List<T>, f: Reducer<T, U>, memo: U): U => {
   if (xs === null) {
       return memo;
   } else {
       return reduce(rest(xs), f, f(memo, first(xs)));
   }
};
```

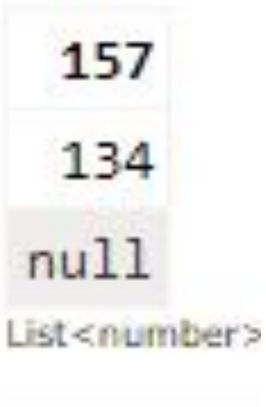
Reducing with sum

```
interface Reducer<T, U> {
   (memo: U, item: T): U;
}

let reduce = <T, U>(xs: List<T>, f: Reducer<T, U>, memo: U): U => {
   if (xs === null) {
       return memo;
   } else {
       return reduce(rest(xs), f, f(memo, first(xs)));
   }
};

let sum = (x: number, y: number): number => {
   return x + y;
};

let totalTime: number = reduce(runTimes, sum, 0);
print(totalTime);
```

What happens when we call reduce on runTimes?



reduce(runTimes, sum, 0)



157
134
null

List<number>

sum(memo, first(runTimes))

$$\text{sum(0, 157)} \longrightarrow 157$$

$$\text{sum(157,134)} \longrightarrow 291$$

When we reach a null we stop!