In [1]:
```
import { requireCytoscape, requireCarbon } from "./lib/draw";

requireCytoscape();
requireCarbon();
```

First-Class Functions

Where Were We?

1. **Language primitives** (i.e., building blocks of languages)
   - Last time: pure functions
   - This time: **first-class functions**
2. Language paradigms (i.e., combinations of language primitives)
3. Building a language (i.e., designing your own language)

Goal

1. Get comfortable with the idea of a *first-class function* (also called *higher-order function*)
2. Learn about first-class functions on arrays (e.g., map, filter, reduce)

Outline

- Why first-class functions?
- First-class functions by example.
- First-class functions on arrays, i.e., map-filter-reduce.

Why First-Class Functions?

Consider the following problem

Problem:

- I have an array of integer numbers (e.g., 1-5 star ratings)
- I want the average of the numbers that are not 1 star ratings (i.e., remove extremely negative reviews)

In [2]:
```
const arr = [1, 2, 3, 4, 5, 2, 2, 1, 1];
```

Let's try an iterative solution first

In [3]:
```
function iterAvgWithout1(arr: number[]): number {
    let [sum, cnt]: [number, number] = [0, 0];  // Question: array or tuple?
    for (const x of arr) {
        if (x > 1) {  // Remove the 1star ratings
            sum += x;
            cnt += 1;
        }
    }
    // TS syntax: === tests equality taking types into account
    // TS syntax: e1 ? e2 : e3 means e2 if e1 is true and e3 otherwise. It i
    return cnt === 0 ? 0 : sum / cnt;
}
```

In [4]:
```
console.log(arr);
iterAvgWithout1(arr);
```

```
[
  1, 2, 3, 4, 5,
  2, 2, 1, 1
]
3
```

In [5]:
```
// Let's add some extra debugging information
function iterAvgWithout1(arr: number[]): number {
    let [sum, cnt, iter]: [number, number, number] = [0, 0, 0];
    for (const x of arr) {
        if (x > 1) {  // Remove the 1star ratings
            sum += x;
            cnt += 1;
        }
        // Purely for illustrative purposes
        console.log(`iter: ${iter} 0th element: ${x}  sum: ${sum}   cnt: ${c
    }
    return cnt === 0 ? 0 : sum / cnt;
}
```

Let's try to filter by 1 and 2 star ratings

In [6]:
```
function iterAvgWithout1And2(arr: number[]): number {
    let [sum, cnt, iter] = [0, 0, 0];
    for (const x of arr) {
        if (x > 2) {  // Remove the 1 and 2 star ratings
            sum += x;
            cnt += 1;
        }
        // Purely for illustrative purposes
        console.log(`iter: ${iter} 0th element: ${x}  sum: ${sum}   cnt: ${c
    }

    return cnt === 0 ? 0 : sum / cnt;
}
```

In [7]:
```
console.log(arr);
iterAvgWithout1And2(arr);
```

```
[
  1, 2, 3, 4, 5,
  2, 2, 1, 1
]
iter: 0 0th element: 1  sum: 0   cnt: 0
iter: 1 0th element: 2  sum: 0   cnt: 0
iter: 2 0th element: 3  sum: 3   cnt: 1
iter: 3 0th element: 4  sum: 7   cnt: 2
iter: 4 0th element: 5  sum: 12   cnt: 3
iter: 5 0th element: 2  sum: 12   cnt: 3
iter: 6 0th element: 2  sum: 12   cnt: 3
iter: 7 0th element: 1  sum: 12   cnt: 3
iter: 8 0th element: 1  sum: 12   cnt: 3
4
```

What just happened?

- We did a copy-paste and changed 1 character ...

Let's try a weighted average

In [8]:
```
function iterWgtAvgWithout1And2(arr: number[]): number {
    let [sum, cnt, iter] = [0, 0, 0];
    for (const x of arr) {
        if (x > 1) {  // Remove the 1 star ratings
            if (x == 2) {
                sum += x;
            } else if (x == 3) {
                sum += 2*x;
            } else {
                sum += 3*x;
            }
            cnt += 1;
        }
        // Purely for illustrative purposes
        console.log(`iter: ${iter} 0th element: ${x}  sum: ${sum}   cnt: ${c
    }
    return cnt === 0 ? 0 : sum / cnt;
}
```

In [9]:
```
console.log(arr);
iterWgtAvgWithout1And2(arr);
```

```
[
  1, 2, 3, 4, 5,
  2, 2, 1, 1
]
iter: 0 0th element: 1  sum: 0   cnt: 0
iter: 1 0th element: 2  sum: 2   cnt: 1
iter: 2 0th element: 3  sum: 8   cnt: 2
iter: 3 0th element: 4  sum: 20   cnt: 3
iter: 4 0th element: 5  sum: 35   cnt: 4
iter: 5 0th element: 2  sum: 37   cnt: 5
iter: 6 0th element: 2  sum: 39   cnt: 6
iter: 7 0th element: 1  sum: 39   cnt: 6
iter: 8 0th element: 1  sum: 39   cnt: 6
6.5
```

Same result ...

- We did another copy-paste and changed the if block
- Say you want to filter out the 1's and 2's, and do a weighted average now ...
- Surely, there must be a better way.

First-Class Functions to the Rescue

- Many programming language features are useful for getting rid of copy-paste. Another reason for a programming language feature is to reduce programmer error (e.g., types).
- First-class functions give us a way to get rid of copy-paste.

Example

Here is an ordinary function.

In [10]:
```
function addOne(x: number): number { // An ordinary function
    return x + 1;
}

addOne(2);
```

```
3
```

Anonymous functions, i.e., a nameless function

- Before we get to first-class functions, it's helpful to introduce the idea of an **anonymous** function.
- An anonymous function is, as it's name suggests, a function without a name.

In [11]:
```
// An anonymous function version of addOne
(x: number) => x + 1
```

```
[Function: tsLastExpr]
```

In [12]:
```
// How to call an anonymous function
((x: number) => x + 1)(2)
```

```
3
```

Because an anonymous function doesn't have a name, we have to copy and paste the code to call it again

In [13]:
```
// How to call an anonymous function again? Copy-paste
((x: number) => x + 1)(2)
```

```
3
```

But what if we could simply assign an anonymous function to an ordinary variable?

In [14]:
```
// 1. (x: number) => number on left-hand side of = is a function type.
// 2. (x: number) => x + 1 on right-hand side of = is an anonymous function.
const f: (foobar: number) => number = (x: number) => x + 1; // 3. naming an
f(2);
```

3

In [15]:
```
const f = (x: number) => x + 1; // Recall that types are optional
f(2);
```

3

In [16]:
```
const g: (x: number, y: number) => number = ((x: number, y: number) => x + y
g(1, 2)
```

3

In [17]:
```
const h: (x: number) => (y: number) => number = (x: number) => (y: number) =
h(1)(2)
```

3

Implications of being able to assign anonymous function to ordinary variable

- We may be used to assigning numbers and strings to variables.
- It is also normal to assign arrays and other data structures (e.g., trees) to variables.
- However, we may never have seen that we can assign a "function" to a variable.
- What are the implications of this feature?
- Your language has first-class functions if you can assign functions to variables.

In [18]:
```
function addOne(x: number): number {
    return x + 1;
}

const f = addOne;  // Assigning a function to variable
f(1);
```

2

1. You can return a function from a function

In [19]:
```
function returnNumberVar(x: number): number {
    const y = x + 1;  // I assign a number to y
    return y;         // And I'm returning y
}

returnNumberVar(2);
```

3

In [20]:
```
function returnFunctionVar(x: number): (whateverIdontcare: number) => number
    const f = (y: number) => x + y;  // I assign a function to f
    return f;                        // And I'm returning f
}

returnFunctionVar(2);
```

```
[Function: f]
```

2. You can pass functions in as arguments

In [21]:
```
const x = 2;          // I assign a number to x
returnNumberVar(x);  // And I'm calling a function with x
```

```
3
```

In [22]:
```
function callFunctionArg(foobar: (y: number) => number): number {  // Return
    return foobar(2);
}

const f = (x: number) => x + 1;  // I assign a function to f
callFunctionArg(f);              // And I'm calling a function with f
```

```
3
```

3. Partially evaluated functions

In [23]:
```
function plus(x: number, y: number): number {
    return x + y;
}

plus(1, 2);
```

```
3
```

In [24]:
```
// anonPlus takes a function, and returns a function
const anonPlus: (x: number) => ((y: number) => number) = (x: number) => ((y:

anonPlus
```

```
[Function: anonPlus]
```

In [25]:
```
const anonPlus1: (y: number) => number = anonPlus(1); // Partial evaluation

anonPlus1
```

```
[Function (anonymous)]
```

In [26]:
```
console.log("evaluating both arguments", anonPlus(1)(2))
console.log("using partially applied function", anonPlus1(2));
```

```
evaluating both arguments 3
using partially applied function 3
```

Returning to motivating problem

- Recall we were trying to filter out low ratings and other various combinations.
- But we found ourselves copy-pasting quite a bit.

In [27]:
```
function fcIterAvgWithFilter(predicate: (x: number) => boolean, arr: number[
    // Passing in a function `predicate` as an argument
    let [sum, cnt, iter] = [0, 0, 0];
    for (const x of arr) {
        if (predicate(x)) {  // Use the predicate
            sum += x;
            cnt += 1;
        }
        // Purely for illustrative purposes
        console.log(`iter: ${iter} 0th element: ${x}  sum: ${sum}   cnt: ${c
    }

    return cnt === 0 ? 0 : sum / cnt;
}
```

In [28]:
```
const filter1 = (x: number) => x > 1;
const filter2 = (x: number) => x > 2;
console.log(fcIterAvgWithFilter(filter1, arr));
console.log(fcIterAvgWithFilter(filter2, arr));
```

```
iter: 0 0th element: 1  sum: 0   cnt: 0
iter: 1 0th element: 2  sum: 2   cnt: 1
iter: 2 0th element: 3  sum: 5   cnt: 2
iter: 3 0th element: 4  sum: 9   cnt: 3
iter: 4 0th element: 5  sum: 14   cnt: 4
iter: 5 0th element: 2  sum: 16   cnt: 5
iter: 6 0th element: 2  sum: 18   cnt: 6
iter: 7 0th element: 1  sum: 18   cnt: 6
iter: 8 0th element: 1  sum: 18   cnt: 6
3
iter: 0 0th element: 1  sum: 0   cnt: 0
iter: 1 0th element: 2  sum: 0   cnt: 0
iter: 2 0th element: 3  sum: 3   cnt: 1
iter: 3 0th element: 4  sum: 7   cnt: 2
iter: 4 0th element: 5  sum: 12   cnt: 3
iter: 5 0th element: 2  sum: 12   cnt: 3
iter: 6 0th element: 2  sum: 12   cnt: 3
iter: 7 0th element: 1  sum: 12   cnt: 3
iter: 8 0th element: 1  sum: 12   cnt: 3
4
```

It's starting to look better!

- Ok so we no longer need to copy and paste code for changing filtering by 1 and 2.
- What about changing the sum?

In [29]:
```
function fcIterAvgWithFilterFun(predicate: (x: number) => boolean, fun: (x:
    let [sum, cnt, iter] = [0, 0, 0];
    for (const x of arr) {
        if (predicate(x)) {  // Use the predicate
            sum += fun(x);
            cnt += 1;
        }
        // Purely for illustrative purposes
        console.log(`iter: ${iter} 0th element: ${x}  sum: ${sum}   cnt: ${c
    }
    return cnt === 0 ? 0 : sum / cnt;
}
```

In [30]:
```
const identity = (x: number): number => x;
const weight = (x: number) => {
    if (x === 2) {
        return x;
    } else if (x === 3) {
        return 2*x;
    } else {
        return 3*x;
    }
};

console.log(fcIterAvgWithFilterFun(filter1, identity, arr));  // average wit
console.log(fcIterAvgWithFilterFun(filter2, identity, arr));  // average wit
console.log(fcIterAvgWithFilterFun(filter1, weight, arr));    // weighted av
console.log(fcIterAvgWithFilterFun(filter2, weight, arr));    // weighted av
```

```
iter: 0 0th element: 1  sum: 0   cnt: 0
iter: 1 0th element: 2  sum: 2   cnt: 1
iter: 2 0th element: 3  sum: 5   cnt: 2
iter: 3 0th element: 4  sum: 9   cnt: 3
iter: 4 0th element: 5  sum: 14   cnt: 4
iter: 5 0th element: 2  sum: 16   cnt: 5
iter: 6 0th element: 2  sum: 18   cnt: 6
iter: 7 0th element: 1  sum: 18   cnt: 6
iter: 8 0th element: 1  sum: 18   cnt: 6
3
iter: 0 0th element: 1  sum: 0   cnt: 0
iter: 1 0th element: 2  sum: 0   cnt: 0
iter: 2 0th element: 3  sum: 3   cnt: 1
iter: 3 0th element: 4  sum: 7   cnt: 2
iter: 4 0th element: 5  sum: 12   cnt: 3
iter: 5 0th element: 2  sum: 12   cnt: 3
iter: 6 0th element: 2  sum: 12   cnt: 3
iter: 7 0th element: 1  sum: 12   cnt: 3
iter: 8 0th element: 1  sum: 12   cnt: 3
4
iter: 0 0th element: 1  sum: 0   cnt: 0
iter: 1 0th element: 2  sum: 2   cnt: 1
iter: 2 0th element: 3  sum: 8   cnt: 2
iter: 3 0th element: 4  sum: 20   cnt: 3
iter: 4 0th element: 5  sum: 35   cnt: 4
iter: 5 0th element: 2  sum: 37   cnt: 5
iter: 6 0th element: 2  sum: 39   cnt: 6
iter: 7 0th element: 1  sum: 39   cnt: 6
iter: 8 0th element: 1  sum: 39   cnt: 6
6.5
iter: 0 0th element: 1  sum: 0   cnt: 0
iter: 1 0th element: 2  sum: 0   cnt: 0
iter: 2 0th element: 3  sum: 6   cnt: 1
iter: 3 0th element: 4  sum: 18   cnt: 2
iter: 4 0th element: 5  sum: 33   cnt: 3
iter: 5 0th element: 2  sum: 33   cnt: 3
iter: 6 0th element: 2  sum: 33   cnt: 3
iter: 7 0th element: 1  sum: 33   cnt: 3
iter: 8 0th element: 1  sum: 33   cnt: 3
11
```

Map-Filter-Reduce Pattern on Arrays

- Every now and then, there exists a pattern that is pretty common such as filtering and performing some function on it
- We want to abstract that out so that a library designer can implement it. This is less work for us, reduces bugs, and introduces opportunities for optimization.

Filter

Take an array and produce an array with some elements removed.

In [31]:
```
function arrFilter<T>(f: (elem: T) => boolean, arr: T[]): T[] { // <T> is ge
    const acc = [];
    for (const x of arr) {
        if (f(x)) {
            acc.push(x);
        }
    }
    return acc;
}
```

In [32]:
```
console.log(arr);
console.log(arrFilter((x: number) => x > 1, arr));
```

```
[
  1, 2, 3, 4, 5,
  2, 2, 1, 1
]
[ 2, 3, 4, 5, 2, 2 ]
```

Map

Take an array and apply a function to each element of that arr.

In [33]:
```
function arrMap<T, U>(f: (elem: T) => U, arr: T[]): U[] {
    const acc = [];  // Create a new array
    for (const x of arr) {
        acc.push(f(x));
    }
    return acc;
}
```

In [34]:
```
console.log("input", arr);
console.log("mapped output", arrMap(weight, arr));
console.log("original input", arr);
```

```
input [
  1, 2, 3, 4, 5,
  2, 2, 1, 1
]
mapped output [
  3, 2, 6, 12, 15,
  2, 2, 3,  3
]
original input [
  1, 2, 3, 4, 5,
  2, 2, 1, 1
]
```

Reduce

Take an array and combine all elements in that array somehow.

In [35]:
```
function arrReduce<T, U>(f: (elem: T, acc: U) => U, initial: U, arr: T[]): U
    let acc = initial;
    for (const x of arr) {
        acc = f(x, acc);
    }
    return acc;
}
```

In [36]:
```
console.log("input", arr);
const arr1 = arrFilter(filter1, arr);
console.log("filtered array", arr1);
const arr2 = arrMap(weight, arr1);
console.log("mapped array", arr2);
console.log("reduce", arrReduce((elem: number, acc: number) => elem + acc, 0
```

```
input [
  1, 2, 3, 4, 5,
  2, 2, 1, 1
]
filtered array [ 2, 3, 4, 5, 2, 2 ]
mapped array [ 2, 6, 12, 15, 2, 2 ]
reduce 39
```

Back to the Original Problem

- Now we'll see how to do the original problem and it's variations using map, filter, and reduce.

In [37]:
```
function fcIterAvgWithMapReduce(pred: (x: number) => boolean, fun: (x: numbe
    const arr2 = arrMap(fun, arrFilter(pred, arr));
    const cnt = arr2.length;
    const sum = arrReduce((elem: number, acc: number) => elem + acc, 0, arr2
    return sum / cnt;
}
```

In [38]:
```
console.log(fcIterAvgWithMapReduce(filter1, identity, arr));
console.log(fcIterAvgWithMapReduce(filter2, identity, arr));
console.log(fcIterAvgWithMapReduce(filter1, weight, arr));
console.log(fcIterAvgWithMapReduce(filter2, weight, arr));
```

```
3
4
6.5
11
```

Shouldn't TypeScript have this functionality for arrays?

Map on arrays

In [39]:
```
console.log("input", arr);
console.log("mapped output", arr.map((x: number) => x + 1));
console.log("original input", arr);
```

```
input [
  1, 2, 3, 4, 5,
  2, 2, 1, 1
]
mapped output [
  2, 3, 4, 5, 6,
  3, 3, 2, 2
]
original input [
  1, 2, 3, 4, 5,
  2, 2, 1, 1
]
```

Filter on arrays

In [40]:
```
console.log("input", arr);
console.log("filtered output", arr.filter((x: number) => x > 1));
console.log("original input", arr);
```

```
input [
  1, 2, 3, 4, 5,
  2, 2, 1, 1
]
filtered output [ 2, 3, 4, 5, 2, 2 ]
original input [
  1, 2, 3, 4, 5,
  2, 2, 1, 1
]
```

Reduce on arrays

In [41]:
```
console.log("input", arr);
const arr1 = arr.filter(filter1);
console.log("filtered arr", arr1);
const arr2 = arr1.flatMap(weight);
console.log("mapped array", arr2);
console.log("reduce", arr2.reduce((elem: number, acc: number) => elem + acc,
```

```
input [
  1, 2, 3, 4, 5,
  2, 2, 1, 1
]
filtered arr [ 2, 3, 4, 5, 2, 2 ]
mapped array [ 2, 6, 12, 15, 2, 2 ]
reduce 39
```

Summary

1. We tried to write a function that does some operations on a collection.
2. We saw that we could use a first-class function to help with the problem of copy-paste when those operations change.
   - You can assign functions to ordinary variables.
   - You can pass functions in as function arguments
   - You can return functions as values from functions.
   - Fun fact: you can encode recursion if you have first-class functions
3. Map/filter/reduce are examples of first-class functions that operate on arrays.

In [ ]: