In [ ]:

```
# import { requireCarbon, requireCytoscape, linePlot } from "./lib/draw";

requireCarbon();
requireCytoscape();
```

Pure Functions

Where Were We?

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

Goal

1. Introduce the concept of a **pure** function.
2. Learn to distinguish between **immutable** (pure) and **mutable** (impure) code.
3. Learn tradeoffs between pure and impure code.

Outline

- What is a pure function?
- Why pure functions?
- Why not pure functions?

What is a Pure Function?

We'll introduce the idea of a pure function via examples.

Example 1

In [2]:

```
function addOne(x: number): number {  // Question: pure or impure?
    return x + 1;
}
```

Pure Functions

1. `addOne` is an example of an **pure** function. A **pure** function is one that produces the same outputs given the same inputs.
2. One constraint that this forces on pure functions is that it cannot modify its input arguments. Otherwise, a pure function might produce different outputs given the same inputs.
3. A pure function must return a value, and hence, its return type is not void.

Same output given same input

In [3]:

```
console.log(addOne(1)); // gives 2 on input 1
console.log(addOne(1)); // gives 2 on input 1
console.log(addOne(1)); // gives 2 on input 1
```

```
2
2
2
```

Does not modify argument

In [4]:

```
const x = 3;
console.log("before", x);  // 3 before function call
console.log("addOne", addOne(x));
console.log("after", x);  // 3 after function call
```

```
before 3
addOne 4
after 3
```

In [5]:

```
let x = 3;
console.log("before", x);  // 3 before function call
console.log("addOne", addOne(x));
console.log("after", x);  // 3 after function call
```

```
before 3
addOne 4
after 3
```

Example 2

In [6]:

```
let y = 1;
function magicAdd(x: number): number {  // Question: pure or impure?
    y += 1;
    return x + y;
}
```

Impure Functions

1. `magicAdd` is an example of an **impure** function. If a function is not a pure function, it is called **impure**. An impure function can produce different outputs given the same inputs. Consequently, it is not a mathematical function.
2. There are at least two ways to create an impure function.
    - **mutate** or change its input arguments. We'll see an example of this later
    - Use global variables. For example, `magicAdd` uses the global variable `y`.
3. The consequence of using impure functions is that calling an impure function multiple times with the same argument may produce different results. For example, calling `magicAdd(1);` produced different results each time.

Note that it does not modify arguments

In [7]:

```
const x = 3;
console.log("before", x);  // 3 before function call
console.log("addOne", magicAdd(x));
console.log("after", x);  // 3 after function call
```

```
before 3
addOne 5
after 3
```

However it gives different outputs given same input

In [8]:

```
console.log(magicAdd(1));
console.log(magicAdd(1));  // gives different output given same input
```

```
4
5
```

Example 3

Suppose you want to concatenate some arrays together.

In [9]:

```
const arr1: number[] = [1, 2, 3];
const arr2: number[] = [4, 5, 6];
```

In [10]:

```
function sensibleConcat(arr1: number[], arr2: number[]): void {  // void mea
    for (const x of arr2) {  // Loop through arr2
        arr1.push(x);  // Add the element x of arr2 to end of arr1
    }
    // Notice no return statement
}
```

Does not return anything and so must mutate arguments

In [11]:
```
sensibleConcat(arr1, arr2);
console.log(arr1);  // Note that arr1 is const, but it is still mutated!
```

```
[ 1, 2, 3, 4, 5, 6 ]
```

Consequently it modifies arguments and is not pure

In [12]:
```
console.log("Before", arr1);
sensibleConcat(arr1, arr2);
console.log("After", arr1);  // Notice that arr1 changes
```

```
Before [ 1, 2, 3, 4, 5, 6 ]
After [
  1, 2, 3, 4, 5,
  6, 4, 5, 6
]
```

And calling the same function multiple times keeps mutating its arguments

In [13]:
```
console.log("Before", arr1);
sensibleConcat(arr1, arr2);
console.log("After", arr1);  // Notice that arr1 changes again
```

```
Before [
  1, 2, 3, 4, 5,
  6, 4, 5, 6
]
After [
  1, 2, 3, 4, 5,
  6, 4, 5, 6, 4,
  5, 6
]
```

PLEASE DO NOT write array concatenation this way

- `sensibleConcat` is an impure function that mutates its input argument `arr1`.
- This is a way to write array concatenation. We will try to convince you later not to do this.

Example 4

In [14]:

```
const arr = []; // NOOOOOOOOO. Don't do it.
function temptingConcat(arr1: number[], arr2: number[]): number[] {  // Ques
    for (const x of arr1) {
        arr.push(x);
    }
    for (const x of arr2) {
        arr.push(x);
    }
    return arr;
}
```

Returns an array and so it does not need to modify arguments

In [15]:

```
const arr1 = [1, 2, 3];  // Notice that we need to reset arr1 because we mut
const arr2 = [4, 5, 6];  // This is optional because we did not mutate arr2.
```

In [16]:

```
console.log("Before", arr1);
const arr3 = temptingConcat(arr1, arr2);  // Notice that we store the result
console.log("After", arr1);  // Notice that arr1 did not change
console.log("Concatenated", arr3);  // Result is in arr3
```

```
Before [ 1, 2, 3 ]
After [ 1, 2, 3 ]
Concatenated [ 1, 2, 3, 4, 5, 6 ]
```

But this gives different outputs for the same input

In [17]:

```
console.log("Before", arr1);
const arr4 = temptingConcat(arr1, arr2);  // Notice that we store the result
console.log("After", arr1);  // Notice that arr1 did not change
console.log("Concatenated", arr3);  // Result is in arr3
```

```
Before [ 1, 2, 3 ]
After [ 1, 2, 3 ]
Concatenated [
  1, 2, 3, 4, 5,
  6, 1, 2, 3, 4,
  5, 6
]
```

PLEASE DO NOT write array concatenation this way

- `temptingConcat` is an impure function that uses a global variable.
- This is a terrible way to write array concatenation.

NO to Example 4. No to temptingConCat Do not. No. NOOOOO

In [18]:

```
function printStuffOut(arr: number[]): number[] {
    for (const x of arr) {
        console.log(x); // Impure function because it prints stuff out
    }
    return arr;
}

printStuffOut([1, 2, 3]);
```

```
1
2
3
[ 1, 2, 3 ]
```

In [19]:

```
function foobar(arr: number[]): number[] {
    for (let i = 0; i < arr.length; i++) {
        arr[i] += 1;
    }
    for (let i = 0; i < arr.length; i++) {
        arr[i] -= 1;
    }
    return arr;
}

console.log(foobar([1, 2, 3]));
console.log(foobar([1, 2, 3]));
console.log(foobar([1, 2, 3]));
```

```
[ 1, 2, 3 ]
[ 1, 2, 3 ]
[ 1, 2, 3 ]
```

Example 5

In [20]:

```
const arr1 = [1, 2, 3];  // Notice that we need to reset arr1 because we mut
const arr2 = [4, 5, 6];  // This is optional because we did not mutate arr2.
```

In [21]:

```
function strangeConcat(arr1: number[], arr2: number[]): number[] {  // we re
    const arr = [];  // Create a new array
    for (const x of arr1) {  // Loop through arr1. Notice that we did not ha
        arr.push(x);  // Add the element x of arr2 to end of arr
    }
    for (const x of arr2) {  // Loop through arr2
        arr.push(x);  // Add the element x of arr2 to end of arr
    }
    return arr;  // Notice return statement
}
```

Returns an array and so it does not need to modify arguments

In [22]:

```
console.log("Before", arr1);
const arr3 = strangeConcat(arr1, arr2);  // Notice that we store the result
console.log("After", arr1);  // Notice that arr1 did not change
console.log("Concatenated", arr3);  // Result is in arr3
```

```
Before [ 1, 2, 3 ]
After [ 1, 2, 3 ]
Concatenated [ 1, 2, 3, 4, 5, 6 ]
```

Same outputs for the same inputs

In [23]:

```
const arr4 = strangeConcat(arr1, arr2);  // Calling strangeConcat twice resu
console.log("Concatenated first time", arr3);
console.log("Concatenated second time", arr4);
```

```
Concatenated first time [ 1, 2, 3, 4, 5, 6 ]
Concatenated second time [ 1, 2, 3, 4, 5, 6 ]
```

It does not matter how many times or how strangeConcat is called

In [24]:

```
console.log("Before", arr1);
const arr5 = strangeConcat(arr3, arr2);  // Notice that we store the result
console.log("After", arr1);  // Notice that arr1 did not change
console.log("Concatenated", arr4);  // Result is in arr4
```

```
Before [ 1, 2, 3 ]
After [ 1, 2, 3 ]
Concatenated [ 1, 2, 3, 4, 5, 6 ]
```

PLEASE write functions in this way

- `strangeConcat` is a pure function
- This may feel like a strange way to write array concatenation. For example, we have to write an extra loop. We will try to convince you that this is a better way to write code.

In [25]:

```
console.log("Our pure concat", strangeConcat(arr1, arr2))
console.log("TypeScript builtin concat", arr1.concat(arr2))
```

```
Our pure concat [ 1, 2, 3, 4, 5, 6 ]
TypeScript builtin concat [ 1, 2, 3, 4, 5, 6 ]
```

Pure Functions and Side-Efffects

Pure functions are also **side-effect** free. Side-effects include:

- Printing stuff to the console.
- Modifying global variables.
- Throwing exceptions.

In [26]:

```
function addOneImpure(x: number): number {
    console.log(x);  // Not pure because printing stuff.
    return x + 1;
}
```

Why Pure Functions?

- The main benefit of pure functions is that it makes your code easier for others to use.
- Because software is primarily developed by teams of people, pure functions give guarantees to your callees that your code won't clobber data structures

Example 1

Consider the following example where Developer 2 is trying to use Developer 1's code, Developer 2 is trying to expose a pure function, and Developer 1 does not guarantee a pure function.

In [27]:

```
function badCodeWrittenByDeveloper1(arr: number[]): void {
    for (let i = 0; i < arr.length; i++) {
        arr[i] += 1;
    }
}
```

In [28]:

```
function defensiveCodeWrittenByDeveloper2(arr: number[]): number[] {
    // I guarantee to my caller that I am a pure function

    // To compensate for the potential of impure code, I have to perform a c
    const arrCopy = [];
    for (const x of arr) {
        arrCopy.push(x);
    }
    badCodeWrittenByDeveloper1(arrCopy);  // Now I can call the code with a
    // Imagine if you had to do this for every library function that you use

    return arrCopy;
}
```

We can fix the code above by having all developers write pure functions.

In [29]:

```
function codeWrittenByDeveloper1(arr: number[]): number[] {
    const ans = [];
    for (const x of arr) {
        ans.push(x + 1);
    }
    return ans;
}
```

In [30]:

```
function codeWrittenByDeveloper2(arr: number[]): number[] {
    // I guarantee to my caller that I am a pure function
    return codeWrittenByDeveloper1(arr);
}
```

Example 2

In [31]:

```
let magic = false
function terribleCodeWrittenByDeveloper1(arr: number[]): void {
    if (magic) {
        for (let i = 0; i < arr.length; i++) {
            arr[i] += 1;
        }
    } else {
        for (let i = 0; i < arr.length; i++) {
            arr[i] += 2;
        }
    }
}
```

In [32]:

```
function terribleCodeWrittenByDeveloper2(x: number): number {
    // All you told me to do was write a function that adds 1
    // And this code compiles and doesn't affect my specification
    magic = !magic;
    return x + 1;
}
```

Why Impure Functions?

- Performance: sometimes writing programs with pure functions, i.e., mutation is too slow.
- Tradeoff: it is harder to reason about what our code is doing. This increases the chance to introduce bugs.
- One strategy might be write a program without state first. If it is too slow, then you can write a version that does use state.

Performance

In [33]:

```
function testMutableConcat(arr: number[], count: number): number[] {
    const tmp: number[] = [];
    // Add arr count times.
    for (let i = 0; i < count; i++) {
        sensibleConcat(tmp, arr);  // Recall this was the mutable version
    }
    return tmp;
}
```

In [34]:

```
function testImmutableConcat(arr: number[], count: number): number[] {
    let tmp: number[] = [];
    for (let i = 0; i < count; i++) {
        tmp = tmp.concat(arr);  // TypeScript's built-in pure version
    }
    return tmp;
}

// 0: [] 0
// 1: [1, 2, 3] 3
// 2: [1, 2, 3] + [1, 2, 3] 6
// 3: [1, 2, 3] + [1, 2, 3] + [1, 2, 3] 9
```

In [35]:

```
function timeFunction(name, f) {
    console.log(`--------------------------`);
    console.log(`${name} started..`);
    const t0 = process.hrtime()
    f();
    const t1 = process.hrtime(t0);
    console.log(`${f.name} completed..`);
    console.info('Execution time (hr): %ds %dms', t1[0], t1[1] / 1000000);
    return t1[0] + t1[1] / 1000000 / 1000;
}

const count = 1000;
timeFunction("Mutable", () => testMutableConcat(Array(100).fill((x) => 0), c
timeFunction("Immutable", () => testImmutableConcat(Array(100).fill((x) => 0
```

```
--------------------------
Mutable started..
 completed..
Execution time (hr): 0s 5.013833ms
--------------------------
Immutable started..
 completed..
Execution time (hr): 0s 76.524459ms
0.07652445899999999
```

In [36]:

```
const counts = [1000, 2000, 4000, 8000];
const mutableTimes = [];
for (const count of counts) {
    let arr = Array(100).fill((x) => 0);
    mutableTimes.push(timeFunction("Mutable", () => testMutableConcat(arr1,
}
const immutableTimes = [];
for (const count of counts) {
    let arr = Array(100).fill((x) => 0);
    immutableTimes.push(timeFunction("Immutable", () => testImmutableConcat(
}

console.log(mutableTimes);
console.log(immutableTimes);
```

```
--------------------------
Mutable started..
 completed..
Execution time (hr): 0s 0.335542ms
--------------------------
Mutable started..
 completed..
Execution time (hr): 0s 0.769209ms
--------------------------
Mutable started..
 completed..
Execution time (hr): 0s 0.57275ms
--------------------------
Mutable started..
 completed..
Execution time (hr): 0s 0.691667ms
--------------------------
Immutable started..
 completed..
Execution time (hr): 0s 1.910792ms
--------------------------
Immutable started..
 completed..
Execution time (hr): 0s 4.240584ms
--------------------------
Immutable started..
 completed..
Execution time (hr): 0s 13.579458ms
--------------------------
Immutable started..
 completed..
Execution time (hr): 0s 100.7835ms
[
  0.000335542,
  0.0007692090000000001,
  0.00057275,
  0.0006916670000000001
]
[ 0.0019107920000000001, 0.004240584, 0.013579458, 0.1007835 ]
```

In [37]:

```
linePlot(counts, [mutableTimes, immutableTimes])
```

Computational complexity detour

- O(N) vs. O(N^2)
- Question: why don't the empirical graphs line up with the theoretical graphs?

Impure functions and time

- Another reason to use impure functions is to encode a notion of **time**: this event should happen before that event.
- This happens in **concurrent** and **distributed** programming.
- We will cover this later.

Summary

1. We learned about pure functions and impure functions.
2. The benefit of using pure functions it that it makes your code easier to reason about. The drawback is that there may be a performance penalty.
3. A good strategy is to write a pure function first. If it is too slow, you can always optimize it by making use of mutation.
4. Most langauges such as TypeScript provide pure functions and optimized implementations

```
In [ ]:
```