3.2 Closures: State and First-Class Functions

In [1]:
```
import ts from "typescript";
import { drawList, requireCytoscape} from "./lib/draw";
import * as introspect from "./lib/introspect";
import { _List, List, Nil, Cons, arrToList, length } from "./lib/list";
import * as list from "./lib/list";
import * as tslab from "tslab";


requireCytoscape();
```

In [2]:
```
function pureFn(x: number): number {
    return x + 1;
}

// f(x) = x^2
// f(1) = 1^2
// f(2) = 2^2
// Same input, same output
```

In [3]:
```
let cnt = 0;
function impureFn(x: number): number {
    cnt += 1;
    return x + cnt;
}

function impureFn2(x: number): number {
    cnt += 1;
    return x + 2* cnt;
}

// f(1!) !+ f(1)

console.log(impureFn(1));
console.log(impureFn(1));
console.log(impureFn(1));
```

```
2
3
4
```

In [4]:
```
function pureArrConcat(arr: number[], arr2: number[]): number[] {
    return arr.concat(arr2);
}
```

In [5]:
```
function impureArrConcat(arr: number[], arr2: number[], arr3: number[]): void {
    for (const x of arr) {
        arr3.push(x);
    }
    for (const x of arr2) {
        arr3.push(x);
    }
}
```

Goal

1. Learn about **closures** and how they capture the idea of state in first-class functions.
2. Get comfortable with using closures.

Outline

- Why closures? (20 min.)
- What exactly is a closure: environment dictionary + code (20 min.)
- Using closures

Closures!?

- We saw **state/references** and **first-class functions**.
- Closures are the link between the two.

Consider the following series of examples

Example 1

A function whose parameter is the same as another variable in scope.

In [6]:
```
const x = 1;
function f(x: number): number {
    return x + 1; // question: what does this x refer to?
}
```

In [7]:
```
f(2) // Question: does this return 3 or 2?
```

3

- The function parameter refers to the closest variable in the text. This is called **lexical scoping**.
- When two variables are named the same, we say that the closest variable in the text **shadows** the further variable.

In [8]:
```
// The equivalent code
const x1 = 1
function f(x2: number): number {
    return x2 + 1;
}
```

Example 2

A function within a function with the same variable names.

In [9]:
```
const x = 1
function f(x: number): number {
    function g(x: number): number {
        return x * 2;  // <- g's
    }
    return g(x + 1); // <- f's
}
f(3)
```

8

In [10]:
```
const x = 1
function f(x: number): number {
    const g: (x: number) => number = ((x: number): number => x * 2)
    return g(x+1)
}
f(3) // Question: what does this return?
```

8

In [11]:
```
const x = 1
function f(x: number): number {
    return ((x: number): number => x * 2)(x+1)
}
```

In [12]:
```
f(3) // Question: what does this return?
```

8

In [13]:
```
// The equivalent code
const x1 = 1
function f(x2: number): number {
    function g(x3: number): number {
        return x3 * 2;
    }
    return g(x2 + 1);
}

f(3)
```

8

Example 3

Returning a function with the same variable names.

In [14]:
```
const x = 1
function f(x: number): (x: number) => number {
```

```
    return (x: number) => x + 2;
}
```

In [15]:
```
f(1) // Question: what does this return?
```

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

In [16]:
```
f(1)(2) // Question: what does this return?
```

```
4
```

In [17]:
```
// The equivalent code
const x1 = 1
function f(x2: number): (x4: number) => number {
    return (x3: number) => x3 + 2;
}

f(1)(2)
```

```
4
```

Example 4

Defining an inner function with the same parameter names.

In [18]:
```
const y = 3;
function f(x: number): number {
    function g(x: number, y: number) {
        return x + y;  // <- which one does this refer to
    }
    return g(x, 1);
}
```

In [19]:
```
f(1) // Question: what does this return?
```

```
2
```

In [20]:
```
// The equivalent code
const y1 = 3;
function f(x1: number): number {
    function g(x2: number, y2: number) {
        return x2 + y2;
    }
    return g(x1, 1);
}
```

Example 5

Our first example of a **closure**.

In [21]:
```
const y = 3;
function f(x: number): number {
    return x + y; // Question: what does this y refer to? Is it undefined?
}
```

In [22]:
```
f(1) // Question: what does this return?
```

```
4
```

In [23]:
```
// Equivalent code
const y = 3;
// Create an environment that saves all the variables referred to in f's body
const env = { "y": y };
function f(env: { [id: string]: any }, x: number): number {
    return x + env["y"];
}
f(env, 1)
```

```
4
```

- The function `f` is said to **capture** the variable `y`. That functions can capture variables should remind you of **references**. It let's you get a handle on things in scope.

A **closure**:

1. Is some code, i.e., a function

2. And an environment, i.e., a dictionary that tells the function how to interpret all the variables that are not in it's local scope.

In [24]:
```
// 5 min, "interesting"
type Dict = (key: string) => any

function set(dict: Dict, key: string, val: any): Dict {
    return (key2: string) => {
        if (key === key2) {
            return val;
        } else {
            return dict(key2);
        }
    }
}

function get(dict: Dict, key: string): any {
    return dict(key);
}

function empty(): Dict {
    return (key: string) => undefined
}

const d1 = empty()
const d2 = set(d1, "x", 1)
const d3 = set(d2, "y", 2)
const d4 = set(d3, "x", 3)
get(d4, "x")
```

```
3
```

Example 6

Another closure.

In [25]:
```
const y = 3;
const z = [1, 2];
function f(x: number): number {
    return x + y + z[0] + z[1];  // question: what does this z refer to?
}
```

In [26]:
```
f(1) // Question: what does this return?
```

```
7
```

In [27]:
```
// Equivalent code
const y = 3;
const z = [1, 2];
// Create an environment that saves all the variables referred to in f's body
const env = { "y": y, "z": z };
function f(env: { [id: string]: any }, x: number): number {
    return x + env["y"] + env["z"][0] + env["z"][1];
}
f(env, 1)
```

```
7
```

Example 7

A closure within a closure.

In [28]:
```
const y = 3;
const z = [1, 2]
function f(x: number): (x: number) => number {
    const z = [3, 4];
    function g(x: number): number {
        return x + y + z[0] + z[1];  // <- what does this z refer to?
    }
    return g;
}
```

In [29]:
```
f(1)(2)  // Question: what does this return?
```

```
12
```

In [30]:
```
// Equivalent code
const y = 3;
const z1 = [1, 2]
const env1 = { "y": y };
function f(_env1: { [id: string]: any }, x1: number): (x: number) => number {
```

```
    const z2 = [3, 4];
    const env2 = { "z2": z2, "y2": _env1["y"]};
    function g(_env2: { [id: string]: any }, x: number): number {
        return x + _env2["y2"] + _env2["z2"][0] + _env2["z2"][1];
    }
    return (x: number) => g(env2, x);
}
```

In [31]:

```
f(env1, 1)(2)
```

```
12
```

Example 8

Does any of this change if we move from `const` to `let` ?

In [32]:

```
let y = 1;
function f() {
    let y = 2;
    function g(x) {
        return x + y;
    }
    return g;
}

console.log(f()(1));
y = 5;
console.log(f()(1));
y = 6;
console.log(f()(1));
```

```
3
3
3
```

Example 9

Does any of this change if we move from `const` to `let` ?

In [33]:

```
let y = 1;
function f() {
    let y = 2;
    function g(x) {
        return x + y;  // <- what does this y refer to?
    }
    y = 4; // Changing y here
    return g;
}

console.log(f()(1));
y = 5;
console.log(f()(1));
y = 6;
console.log(f()(1));
```

```
5
5
5
```

- The function `g` captures the variable `y = 2` .
- When it changed to `y = 4` , the function g still has access to `y` .

Example 10

In [34]:

```
const arr = [];

for (let i = 0; i < 4; i++) {
    // 4 i's created
    arr.push((x: number) => x + i);
}
```

In [35]:

```
// What is printed?
console.log(arr[0](1));
console.log(arr[1](1));
console.log(arr[2](1));
console.log(arr[3](1));
```

```
1
2
```

```
3
4
```

Example 11

In [36]:
```
const arr = [];
let i = 0; // only one i variable
for (i = 0; i < 4; i++) {
    arr.push((x: number) => x + i);
}
```

In [37]:
```
// What is printed?
console.log(arr[0](1));
console.log(arr[1](1));
console.log(arr[2](1));
console.log(arr[3](1));
```

```
5
5
5
5
```

Example 12

Var vs. let, so don't use let

In [38]:
```
const arr = [];
for (var i = 0; i < 4; i++) {
    arr.push((x: number) => x + i);
}
```

In [39]:
```
console.log(arr[0](1));
console.log(arr[1](1));
console.log(arr[2](1));
console.log(arr[3](1));
```

```
5
5
5
5
```

What's the big deal with Closures?

Application 1: Closures and Encapsulation

- Recall that one of the benefits of Objects and Classes was that we could hide data.

In [40]:
```
class Counter {
    private count: number;

    constructor() {
        this.count = 0;
    }

    increment() {
        this.count += 1;
        return this.count;
    }

    getCount() {
        return this.count;
    }
}
```

In [41]:
```
const counter = new Counter();
console.log(counter.increment());
console.log(counter.increment());
console.log(counter.increment());
```

```
1
2
3
```

In [42]:
```
// This fails at compile-time because count is private
try {
    counter.count += 1;
} catch (e) {
```

```
    console.log(e);
}
```

```
3:13 - Property 'count' is private and only accessible within class 'Counter'.
```

In [43]:
```
console.log(counter.increment());
```

```
4
```

Attempt with functions: Take 1

In [44]:
```
let count = 0;

function increment() {
    count += 1;
    return count;
}

console.log(increment());
console.log(increment());
console.log(increment());
count += 1;
console.log(increment());
```

```
1
2
3
5
```

- The issue is that count is still in scope. We could not hide the variable count.
- What if we wanted two counters?

Attempt with functions: Take 2

In [45]:
```
type FunCounter = {increment: () => void, getCount: () => number};

function makeCounter(): FunCounter {
    let count = 0;

    function _increment() {
        // Increment **captures** the variable count
        count += 1;
        return count;
    }

    function _getCount() {
        return count;
    }

    return {increment: _increment, getCount: _getCount};
}
```

In [46]:
```
const counter = makeCounter();
console.log(counter.increment());
console.log(counter.increment());
console.log(counter.increment());
```

```
1
2
3
```

In [47]:
```
// This also fails at compile-time because count is private
try {
    counter.count += 1;
} catch (e) {
    console.log(e);
}
```

```
3:13 - Property 'count' does not exist on type 'FunCounter'.
```

In [48]:
```
console.log(counter.increment());
console.log(counter.getCount());
```

```
4
4
```

Closure's are kind of like "Objects"

- We just saw how to encode objects with first-class functions.
- Crucially, we needed the ability to close over variables in a function's local scope.

- Challenge: how would we encode something like inheritance?

Application 2: Closures and Callbacks

- Closures are great for implementing **callbacks**: call this function when some event happens.
- Events are commonly things such as user-input (e.g., clicks, typing).
- We are in TypeScript, which is a superset of JavaScript, so we should have access to user-events from the browser.
- Since we are in a Jupyter notebook, we will use our Jupyter notebook's ability to display raw HTML to illustrate this concept first.

In [49]:
```
import * as tslab from "tslab";

tslab.display.html(`
<!-- This part is HTML -->
<p onclick="callback(this)">Click me to change my text color.</p>

<!-- This part is TypeScript -->
<script>
let lastIdx = 0;
const colors = ['red', 'green', 'blue'];

function callback(elmnt) {
    // This HTML element has a style.color property
    // colors refers to the on in lexical scope and will be captured by callback.
    // lastIdx refers to the one in lexical scope and will be captured by callback.
    elmnt.style.color = colors[lastIdx];

    lastIdx = (lastIdx + 1) % (colors.length);
}
</script>
`);
```

Click me to change my text color.

- Pretty cool right?
- But ... we're programming with strings again. This should remind you of all the bad things with copy-and-paste.
  - No syntax highlighting
  - No static checking
  - What if I can't modify the code?
- How can we fix this?

Closures to the rescue!

In [50]:
```
// Let's just package our code inside a function.
function codeBlock() {
    let lastIdx = 0;
    const colors = ['red', 'green', 'blue'];

    function callback(elmnt) {
        // This HTML element has a style.color property
        // colors refers to the on in lexical scope and will be captured by callback.
        // lastIdx refers to the one in lexical scope and will be captured by callback.
        elmnt.style.color = colors[lastIdx];
        lastIdx = (lastIdx + 1) % (colors.length);
    }

    return callback;
}
```

In [51]:
```
function displayHTMLWithCallback(closure) {
    tslab.display.html(`
    <p onclick="callback(this)">Click me to change my text color.</p>

    <script>
    // We won't understand this fully for now, but we are essentially using code to do the copy-paste for us.
    ${closure.toString()}
    // Calling our closure will unpackage the function.
    const callback = ${closure.name}();
    </script>
    `);
}

displayHTMLWithCallback(codeBlock)
```

Click me to change my text color.

Application 3: Closures and Pure Functions

In [52]:
```
function fibonnaci(n: number): number { // this is a pure function
    if (n < 0) {
        throw Error("Positive numbers only");
    }

    let cache: { [id: number]: number } = {}; // <- it uses references + mutation
    function go(n: number): number {  // go is a closure that captures `cache`
        if (n in cache) {
            return cache[n];
        }

        if (n == 0) { // F_0 = 1
            cache[n] = 1;
            return 1;
        } else if (n == 1) { // F_1 = 1
            cache[n] = 1;
            return 1;
        } else { // F_n = F_{n-1} + F_{n-2}
            cache[n-1] = go(n-1);
            cache[n-2] = go(n-2);
            return cache[n-1] + cache[n-2];
        }
    }

    return go(n);
}
```

Summary

- End bottom up portion
- We saw recursion + algebraic data types
- And today we saw closures, i.e., first-class functions + dictionaries (i.e., references / state)

In [ ]: