FUNCTIONAL PROGRAMMING IN JavaScript

CASSIDY WILLIAMS

SOFTWARE ENGINEER AT VENMO
CASSIDY@VENMO.COM

WHAT IS FUNCTIONAL PROGRAMMING?

"The mustachioed hipster of programming paradigms"

SMASHING MAGAZINE

It produces *abstraction* through clever ways of combining functions.

There are two things you need to know to understand functional programming.

FUNCTIONS ARE

Immutable

If you want to change data in an array, just return a new array with the changes, don't change the original!

FUNCTIONS ARE

Stateless

Functions act as if for the first time, every time!

In addition, there are 3 best practices you should follow.

1) Your functions should accept at least 1 argument

2) Your functions should either return data, or another function

3) NO LOOPS

There are languages made for this

- Lisp
- Scheme
- Haskell
- Scala
- Clojure

But we're going to use JavaScript.

LET'S GO THROUGH SOME EXAMPLES.

(WE'RE ABOUT TO CODE, GET YOUR LAPTOPS READY)

Everything we write will be in the same JS file.

Type out these helpers.

```
function log(arg) {
  document.writeln(arg);
}

function identity(x) {
  return x;
}

function add(a, b) {
  return a + b;
}

function sub(a, b) {
  return a - b;
}
```

Recursion is a big deal in functional programming.

Write a function that takes an argument and returns a function that returns that argument.

Write a function that takes an argument and returns a function that returns that argument.

```
function identityf(arg) {
  return function() {
    return arg;
  };
}
```

What the heck does this mean?

```
> identity(5)
5

> identityf(5)
function () {
    return arg;
  }

> identityf(5)()
5
```

Write a function that adds from two invocations.

```
addf(3)(4) // this returns 7.
```

Write a function that adds from two invocations.

```
function addf(x) {
    return function (y) {
        return add(x, y);
    };
}
```

Write a function that takes in a function and an argument, and returns a function that can take a second argument.

```
curry(add, 9)(3) // this adds 9 and 3 together -> returns 12
```

Write a function that takes in a function and an argument, and returns a function that can take a second argument.

```
function curry(fun, a) {
  return function(b) {
    return fun(a, b)
  };
}
```

YOU JUST LEARNED CURRYING!

Currying is when you break down a function that takes multiple arguments into a series of functions that take part of the arguments.

Write a function that takes a binary function and makes it callable with 2 invocations.

```
liftf(add)(2)(3) // this adds 2 and 3 -> returns 5
liftf(sub)(10)(7) // this is 10 - 7 -> returns 3
```

Write a function that takes a binary function and makes it callable with 2 invocations.

```
function liftf(fun) {
  return function(a) {
    return function(b) {
      return fun(a, b);
    };
  };
}
```

So, using the functions we've written so far, write a function `increment` in 2 different ways.

```
var increment = curry(add, 1);

> increment(5)
6
```

Using the functions we've written so far, write a function `increment` in 2 different ways.

```
var increment1 = addf(1);
var increment2 = liftf(add)(1);
```

Write a function that reverses the arguments of a binary function.

```
reverse(sub)(2, 3) // returns sub(3, 2) -> 1
```

Write a function that reverses the arguments of a binary function.

```
function reverse(fun) {
  return function(a, b) {
    return fun(b, a);
  };
}
```

Now let's get funky, and make a function that returns an object.

Write a function `counter` that returns an object containing two functions that implement an up/down counter.

```
> var k = counter(6)
> k.next()
7
> k.next()
8
> k.prev()
7
```

Write a function `counter` that returns an object containing two functions that implement an up/down counter.

```
function counter(arg) {
  return {
    next: function() { return arg += 1; },
    prev: function() { return arg -= 1; }
  };
}
```

Write a function that returns a generator that will return the next fibonacci number.

```
> var t = fibonaccif(0,1)
> t()
0
> t()
1
> t()
1
> t()
2
> t()
3
```

Write a function that returns a generator that will return the next fibonacci number.

```
function fibonaccif(a, b) {
  return function() {
    var n = a;
    a = b;
    b += n;

    return n;
  };
}
```

LAST ONE.

Write a function that adds from many invocations, until it sees an empty invocation.

```
addgroup()                  // returns undefined
addgroup(2)()               // returns 2
addgroup(2)(7)()            // returns 9
addgroup(3)(4)(0)()         // returns 7
addgroup(1)(2)(4)(8)()      // returns 15
```

Write a function that adds from many invocations, until it sees an empty invocation.

```
function addgroup(a) {
  if(a === undefined) return a;
  return function g(b) {
    if(b !== undefined) {
      return addgroup(a+b);
    }
    return a;
  };
}
```

WASN'T THIS

FUN?

WHY DID WE JUST LEARN FUNCTIONAL PROGRAMMING?

- Functions can be broken down into simpler and smaller chunks that are easier to read
- Software is more reliable due to its modularity
- It's becoming more popular EVERY SINGLE DAY.

Helpful Libraries

- fn.js
- underscore.js
- bacon.js

CASSIDY WILLIAMS

CASSIDY@VENMO.COM

HAVE FUN AT *PennApps!*