Getting Started with Functional Programming in JavaScript

Eric Normand

PurelyFunctional.tv

MILK
BIly
MIlk

Buying milk

- Drive to store
- Get shopping basket
- Walk to milk section
- Put milk in basket
- Walk to cashier
- Pay for milk
- Drive home

Making groceries

- Drive to store
- Get shopping basket
- For each item you need
  - Walk to that section
  - Put item in basket
- Walk to cashier
- Pay
- Drive home

Make shopping list

- Open fridge
- Look at contents
- Note down any items that are low/missing
- Close fridge

Making groceries

- Make shopping list
- Drive to store
- Get shopping basket
- For each item on list
  - Walk to that section
  - Put item in basket
- Walk to cashier
- Pay
- Drive home

Diff

- Given what we actually have
- and given what we need
- Write down a list of things we need that we don’t have

Make shopping list

- Open fridge
- Look at contents => what we actually have
- Close fridge
- Diff(what we have, what we need)

Actions

Open fridge

Look

Drive to store

Pay

Calculations

Diff

Pathfinding

Sum total

Data

Shopping list

Map of store

Receipt

Actions

the process of doing something, typically to achieve an aim

- Typically called *Effects* or *Side-effects*
- Depend on *when you run them* or *how many times you run them*

Calculations

computation from inputs to outputs

- Eternal — outside of time
    - doesn’t matter when or how many times
- Opaque
    - don’t know what it does until you run it

Data

factual information used as a basis for reasoning, discussion, or calculation

- Inert
- Self-identical
    - It is what it is
- Requiring interpretation

Implementation

JavaScript

- Data — built-in types
  - Arrays
  - Objects
  - Strings
  - Numbers
- Calculations — pure functions
- Actions — impure functions

Recommendation:
Identify Actions, Calculations,
and Data in your existing code

```
var stuffIBuy = [
  “broccoli”, “milk”, “eggs”, “bread”
];
var shoppingList = [];

function makeShoppingList() {
  fridge.open();
  var contents = fridge.look();
  for(i = 0; i < stuffIBuy.length; i++) {
    if(contents.indexOf(stuffIBuy[i]) < 0) {
      shoppingList.push(stuffIBuy[i]);
    }
  }
  fridge.close();
}
```

```
var stuffIBuy = [
  “broccoli”, “milk”, “eggs”, “bread”
];
var shoppingList = [];

function makeShoppingList() {
  fridge.open();
  var contents = fridge.look();
  for(i = 0; i < stuffIBuy.length; i++) {
    if(contents.indexOf(stuffIBuy[i]) < 0) {
      shoppingList.push(stuffIBuy[i]);
    }
  }
  fridge.close();
}
```

```
var stuffIBuy = [
  “broccoli”, “milk”, “eggs”, “bread”
];
var shoppingList = [];

function makeShoppingList() {
  fridge.open();
  var contents = fridge.look();
  for(var i = 0; i < stuffIBuy.length; i++) {
    if(contents.indexOf(stuffIBuy[i]) < 0) {
      shoppingList.push(stuffIBuy[i]);
    }
  }
  fridge.close();
}
```

Action

Calculation

Data

Recommendation: Avoid global mutable state

```
var stuffIBuy = [
  “broccoli”, “milk”, “eggs”, “bread”
];
var shoppingList = [];

function makeShoppingList() {
  fridge.open();
  var contents = fridge.look();
  for(i = 0; i < stuffIBuy.length; i++) {
    if(contents.indexOf(stuffIBuy[i]) < 0) {
      shoppingList.push(stuffIBuy[i]);
    }
  }
  fridge.close();
}
```

```
var stuffIBuy = [
  “broccoli”, “milk”, “eggs”, “bread”
];

function makeShoppingList() {
  fridge.open();
  var contents = fridge.look();
  var shoppingList = [];
  for(i = 0; i < stuffIBuy.length; i++) {
    if(contents.indexOf(stuffIBuy[i]) < 0) {
      shoppingList.push(stuffIBuy[i]);
    }
  }
  fridge.close();
  return shoppingList;
}
```

Recommendation:
Refactor to separate out Actions from Calculations from Data

```
var stuffIBuy = [
  “broccoli”, “milk”, “eggs”, “bread”
];

function diff(actual, needed) {
  var ret = [];
  for(i = 0; i < needed.length; i++) {
    if(actual.indexOf(needed[i]) < 0) {
      actual.push(needed[i]);
    }
  }
  return ret;
}

function makeShoppingList() {
  fridge.open();
  var contents = fridge.look();
  fridge.close();
  return diff(contents, stuffIBuy);
}
```

```
var stuffIBuy = [
 “broccoli”, “milk”, “eggs”, “bread”
];

function makeShoppingList() {
 fridge.open();
 var contents = fridge.look();
 var shoppingList = [];
 for(i = 0; i < stuffIBuy.length; i++) {
  if(contents.indexOf(stuffIBuy[i]) < 0) {
   shoppingList.push(stuffIBuy[i]);
  }
 }
 fridge.close();
 return shoppingList;
}
```

Calculations

- Much more testable
  - Run whenever you want
  - Run as many times as you want
  - Define exact inputs and check outputs
- More reusable

Data

- Serializable
  - Store to disk
  - Send over the wire
- Usable in multiple contexts

Recommendation:
Create an Action function, create a Calculation function,
and create a “convenience” function that puts them together

What is Functional Programming?

Why use Functional Programming?

paradigm

a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated

Merriam-Webster

philosophical or
theoretical framework,
world view

theories, laws, generalizations

basic assumptions, ways of thinking, methodology

What is Functional Programming?

Why use Functional Programming?

Goals of my Theory

- Explain what it is we (functional programmers) actually do
    - in terms we can all understand
- Explain why it has advantages over other paradigms
    - to people who haven’t done FP
- Avoid focusing on features
- Give explanatory and predictive power
- Self-described functional programmers should agree

My Theory of FP

Actions

Data

Calculations

Actions

the process of doing something, typically to achieve an aim

- Typically called *Effects* or *Side-effects*
- Depend on *when you run them* or *how many times you run them*
- Examples
  - Sending a message over the network
  - Writing to file system — other programs can see the change
  - Changing or reading mutable state

Data

factual information used as a basis for reasoning, discussion, or calculation

- Inert
- Serializable
- Requiring interpretation
- Examples
  - Numbers
  - Bytes
  - Strings
  - Collections

Calculations

computation from inputs to outputs

- Mathematical functions
- Eternal — outside of time
- Referentially transparent
- Examples
  - List concatenation
  - Summing numbers

Contrast with OOP

OOP

Objects

References

Messages

Implementation

Haskell

- Data — built-in types and defined types
- Calculations — functions
- Actions — IO type

Implementation

Clojure

- Data — built-in types
- Calculations — pure functions
- Actions — impure functions

Further down the rabbit hole

- Everything “First-class”
  - Data
  - Calculations
  - Actions
- Minimum necessary to program functionally in a language

Further down the rabbit hole

- Data may represent Calculations
  - [:sum 0 1 2 3 4 5]
- Data may represent Actions
  - [:send “some message”]

Domains are separate

Data

Data + Data => Data

Examples

- Addition
- Concatenation

Calculations

Calc + Calc => Calc

Actions

- Contagious!
  - Calculation + Action => Action
  - Data + Action => Action
  - Examples
    - Print the square of a number — square => print!
    - Parse the input as a number — read! => parse

Calculations

- Algebraic manipulation
- Turing complete
  - implies the Halting problem
- Opaque
  - What is this code going to do?
  - Only way to know is to run it

Data

- Can represent something else
- Structure
  - Known Big-O complexities

Refactorings

Actions

- Action => Action + Calculation
- Action => Action + Data
- Action => Action + Action

Calculations

- Calculation => Calculation + Data
- Calculation => Calculation + Calculation

Calculations can be manipulated algebraically

- Know some properties without running

What counts as an Action?

Calculations	Actions
Timeless	Bound in time
Pure function	Read/write to disk
Pure function takes 24 hours to compute	
	Read/write to temp file as buffer

Actions

how many times they run

always matters - 0≠1≠more

launching a missile

sending an email

idempotent - 0≠1=more

setting public flag to true

free of side-effects - 0=1=more

GET request

reading mutable state

Actions

when they run

transactional read

guaranteed to be consistent

transactional+serialized writes

Order matters, but at least it’s some order

exactly once reads

Communicating Sequential Processes

Eric Normand

LispCast

Follow Eric on:

in **Eric Normand**

@EricNormand

● **lispcast.com**

✉ **eric@lispcast.com**