Explorations in Functional Programming with Javascript

Aaron Lukacsko
Lead Developer, Progressive Insurance
Co-Founder, Ginormous Industry, Inc.

Functional Programming - what is it?

- Declarative Programming
  - “What’ the program must accomplish through expressions
  - Expresses logic of computation without describing control flow
    - SQL
- An alternative paradigm to imperative programming
  - Telling the computer “how” to do work
  - Mutating State
  - Procedural programming
    - COBOL, C, javascript
  - Object Oriented Programming
    - C++, Smalltalk, C#, javascript

Imperative Example - Procedural

```
var albums = require('./random-music.json')

for(var i = 0; i < albums.length; i++) {
  console.log(albums[i].title)
}

// or maybe using a little cleaner syntax
for(album of albums) {
  console.log(album.title)
}
```

Imperative Example - Object-Oriented

```
var albums = require('./random-music.json')

function makeMyAlbum(album) {
  return {
    title: album.title,
    year: album.year,
    play: function() {
      console.log('playing ' + album.title + ' from ' + album.year)
    }
  }
}

var myAlbum = makeMyAlbum(albums[4])
myAlbum.play()

// you can also access those properties
console.log(myAlbum.title)
```

Declarative Example - Functional

```
const albums = require('./random-music.json')
albums.forEach(album => console.log(album.title))
```

Concepts of Functional Programming

- Higher-order functions
  - Functions that take functions as input and return other functions
- Pure functions
  - Rely only on their inputs - same inputs will always give same output
  - No side-effects - no changes to global state
  - Allows for optimizations
    - same inputs yielding same outputs → output is constant
    - Parallel processing possible
- Immutability
  - Data never changes
- Unit of work is the function

Javascript is not (only) a functional language

- Javascript not purely a functional programing language
  - You can mutate data / variables
  - You can write procedurally or object-oriented as well as functional
- ES6 syntax and features have gone a long way to better enabling FP
  - Arrow functions
  - Functional building blocks like map and reduce
- Large OSS ecosystem to fill holes in Javascript as a functional language
  - Libraries on github / npm to install
  - Lodash, underscore, ramda
- Ramda
  - Ramda emphasizes a purer functional style

Examples

- Any recent node.js and npm install
- Get data file from https://pastebin.com/jWEvZCiE
- https://github.com/aalukacsko/functional-explorations
- Data sourced from: https://data.discogs.com
- In a new empty folder:

```
npm init -y
npm i ramda -S
```

- Open your editor in the folder

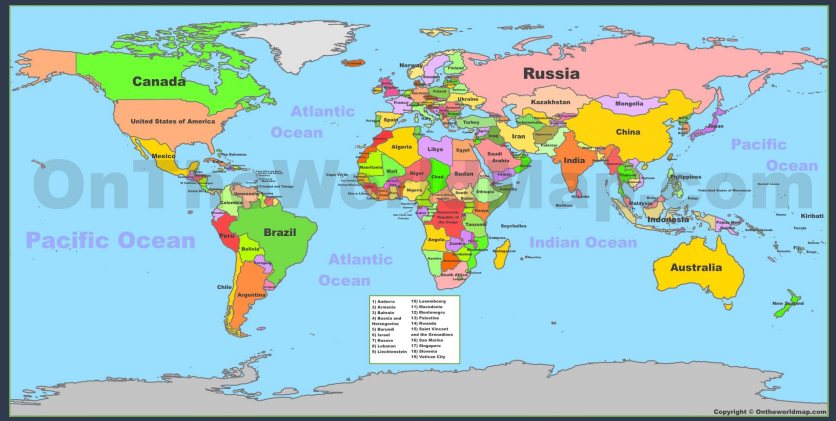
forEach

- Takes a function and a collection and calls the function on each item in the collection

```
R.forEach( album => console.log(album), albums )

const prettyAlbum = album => `${album.title} by ${album.artist} from ${album.year}`
R.forEach( a => console.log(prettyAlbum(a)), albums )
```

map



- Takes a function and a collection and returns a new, transformed collection

```
const prettyAlbumCollection = R.map(album => prettyAlbum(album), albums)
console.log(prettyAlbumCollection)
```

```
[ 'Seviljski Brijač by Tomislav Neralić from 1966 genre: Classical',
  'Pianoconcert Nr. 21 / "Eine Kleine Nachtmusik" by Wolfgang Amadeus Mozart , Dame Moura Lympany from 1972...
  'Recorder Music: Old And New by The Duschenes Recorder Quartet from 1965 genre: Classical',
  'Concerts Of Great Music Age Of Elegance by Ludwig Van Beethoven , Luigi Boccherini , Christoph Willibald...
  'I Only Have Eyes For You by Billy Vaughn from 1967 genre: Classical',
...
```

filter

- Selects items from collection based on filter function

```
const filterYear1964 = a => a.year === 1964
const albumsFrom1964 = R.filter(filterYear1964, pricedAlbums)
```

- Ramda also provides a `reject` function that does the opposite

reduce

- Takes a function with two arguments, an initial value and a collection to operate on

```
const reducer = (totalPrice, album) => totalPrice + album.price
const grandTotal = R.reduce(reducer, 0, pricedAlbums)
```

- Central to the redux design pattern
  - Basic redux form: `(previousState, action) => newState`

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

Redux example from:
https://redux.js.org/basics/reducers

Partial Application

- Applying a function to *some* of its arguments
- Partially applied function is returned waiting for future arguments
- Taking a function and reducing the number of arguments

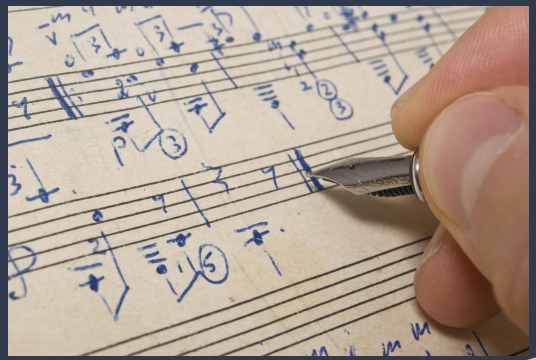
```
const filterYear = (y, a) => a.year === y
const filterYearPartial = y => a=> a.year
=== y
// with ramda you can use their partial
function when an argument is fixed
const filterYear1959 = R.partial(filterYear,
[1959])
```

Currying

- A technique for transforming functions that helps with partial application
- Taking a function and changing it so it takes a single argument and returns a single function

```
const filterYear = (y, a) => a.year === y
const filterYearCurry = R.curry(filterYear)
const filteredAlbums1959 =
R.filter(filterYearCurry(1959), pricedAlbums)
console.log(filteredAlbums1959)
```

Composition



- Combining two or more functions to produce a new function
- Data flows through the ‘pipeline’

```
const pricedAlbums = R.map(setPrice, albums)
const jazzAlbums = R.filter(filterJazz, pricedAlbums)
const albumsFrom1959 = R.filter(filterYearCurry(1959),
jazzAlbums)
const prettyJazzAlbums1959 = R.map(prettyAlbum,
albumsFrom1959)
```

Testing


```
describe('albums', () => {
  const mockAlbums = [
    { title: 'test', year: 1940, artist: 'someone' },
    { title: 'test2', year: 1959, artist: 'somebody' },
    { title: 'test3', year: 1959, artist: 'someoneElse' }
  ]

  const expectedAlbums = [
    { title: 'test2', year: 1959, artist: 'somebody' },
    { title: 'test3', year: 1959, artist: 'someoneElse' }
  ]

  it(
      'should only contain albums from 1959',
      function () {
          expect(albumsFrom1959(mockAlbums)).toBe(expectedAlbums)
      })
})
```

Debugging

```
const log = x => { console.log(x); return x }

const prettyJazzAlbums1959 = R.pipe(
 pricedAlbums_,
 log,
 jazzAlbums,
 log,
 albumsFrom1959,
 log,
 prettyOutput
)
console.log(prettyJazzAlbums1959(albums))
```

Resources

- Ramda: http://ramdajs.com/
- Why Ramda? http://fr.umio.us/why-ramda/
- Mostly adequate guide to FP (in javascript): https://github.com/MostlyAdequate/mostly-adequate-guide
- Awesome FP JS: https://github.com/stoeffel/awesome-fp-js
- Egghead.io: https://egghead.io/courses/professor-frisby-introduces-composable-functional-javascript
- Haskell: http://learnyouahaskell.com/
- Aaron's github: https://github.com/aalukacsko/functional-explorations