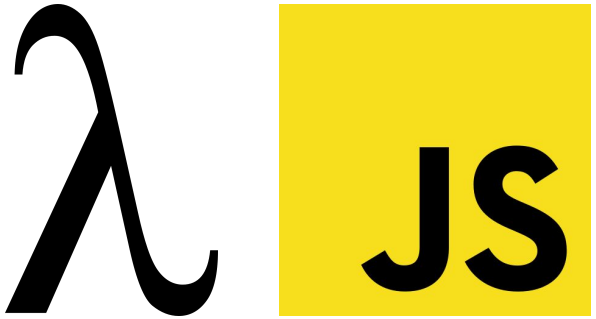


Programação Funcional

& JavaScript

Sobre

Carlos Roberto
Front-End Developer na iClinic

@carlosrberto
medium.com/@carlosrberto
github.com/carlosrberto

Objetivos

- Visão geral sobre programação funcional
- Vantagens e benefícios da programação funcional
- Conceitos do mundo funcional
- Introdução a programação funcional no JavaScript

Cálculo-λ

- Sistema matemático formal criado por **Alonzo Church** em **1936**.

- O **cálculo-λ** pode ser visto como uma linguagem de programação abstrata em que funções podem ser **combinadas** para formar outras funções, de uma forma **pura**.

- O **cálculo-λ** trata funções como **cidadãos de primeira classe**, isto é, entidades que podem, como um dado qualquer, ser utilizadas como argumentos e retornadas como valores de outras funções.

O que é programação funcional?

- Um paradigma de programação
- Outra forma de se pensar para resolver problemas
- Se baseia no uso de **funções matemáticas** para **dividir problemas** em **pequenas partes** e **compor soluções.**
- Sua criação teve origem no **cálculo-λ**

Imperativo vs Funcional

```
let word = "javascript";
let newWord = "";
for(let i = word.length - 1; i >= 0; i--) {
   newWord = newWord + String.fromCharCode(word.charCodeAt(i) & 223);
}
```

```
console.log(newWord) // TPIRCSAVAJ
```

```
(map Data.Char.toUpper . reverse) "haskell"
```

```
"LLEKSAH"
```

```
"javascript".split("").reverse().join("").toUpperCase();
```

```
compose(toUpperCase, reverse)("javascript");
```

Como eu cheguei até a programação funcional?

Como a programação funcional pode nos ajudar?

Se faça as seguintes perguntas:

- Constantemente preciso refatorar o código para adicionar novas funcionalidades? (extensibilidade)
- Se eu altero um arquivo, outro arquivo é afetado? (modularização)
- Existe muito código duplicado? (reusabilidade)
- Testar meu código é muito trabalhoso? (testabilidade)
- Meu código é desorganizado e é difícil entender o seu fluxo?

O que eu preciso fazer para usar programação funcional?

Devo aprender uma linguagem funcional primeiro?

PURESCRIPT

A strongly-typed functional programming language that compiles to JavaScript

BENEFITS

- Compile to readable JavaScript and reuse existing JavaScript code easily
- An extensive collection of libraries for development of web applications, web servers, apps and more
- Excellent tooling and editor support with instant rebuilds
- An active community with many learning resources
- Build real-world applications using functional techniques and expressive types, such as:
  - Algebraic data types and pattern matching
  - Row polymorphism and extensible records
  - Higher kinded types
  - Type classes with functional dependencies
  - Higher-rank polymorphism

HELLO, PURESCRIPT!

```
import Prelude
import Effect.Console (log)

greet :: String -> String
greet name = "Hello, " <> name <> "!"

main = log (greet "World")
```

QUICK START GUIDE

TRY PURESCRIPT

Get the compiler

BINARIES

Precompiled binaries are available for OSX, Linux, and Windows from the latest release page on GitHub.

NPM

```
npm install -g purescript
```

(Installation via `npm` requires Node version 6 or later)

TOOLS

Learn more

LIBRARIES

The Pursuit package database hosts searchable documentation for PureScript packages.

DOCUMENTATION

Visit the documentation repository on GitHub, where you can find articles, tutorials, and more.

BOOK

examples docs community blog

elm

A delightful language for reliable webapps.

Generate JavaScript with great performance and no runtime exceptions.

Try Online Install

Features

JavaScript Interop

Elm compiles to JavaScript, so trying out Elm is easy. Convert a small part of your app to Elm and embed it in JS. No full rewrites, no huge time investment. More about that here.

```
var Elm = require('dist/elm/app.js');
var node = document.getElementById('elm-app');
var app = Elm.App.embed(node);
```

No Runtime Exceptions

Unlike hand-written JavaScript, Elm code does not produce runtime exceptions in practice. Instead, Elm uses type inference to detect problems during compilation and give friendly hints. This way problems never make it to your users. NoRedInk has 80k+ lines of Elm, and after more than a year in production, it still has not produced a single

```
The 1st argument to function `join` is causing a mismatch.

4|   String.join 42 ["Alice","Bob"]
                  ^^
Function `join` is expecting the 1st argument to be:

    String

But it is:

    number
```

Programação funcional no JavaScript

De início podemos aplicar conceitos funcionais para melhorar a qualidade do software que escrevemos.

Algumas limitações da linguagem que podem ser contornadas:

- Imutabilidade
- Checagem de tipo
- Lazy evaluation
- Memoization
- Recursão
- Recursos avançados de programação funcional

flow Getting Started Docs Try Blog

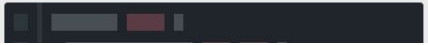
FLOW IS A STATIC TYPE CHECKER FOR JAVASCRIPT.

GET STARTED INSTALL FLOW Star 17,290

Current Version: v0.77.0

CODE FASTER.

Tired of having to run your code to find bugs? Flow identifies problems as you code. Stop wasting your time guessing and checking.



Getting Started

Introduction to type checking with Flow

Getting Started
Installation
Usage

Flow is a static type checker for your JavaScript code. It does a lot of work to make you more productive. Making you code faster, smarter, more confidently, and to a bigger scale.

Flow checks your code for errors through **static type annotations**. These *types* allow you to tell Flow how you want your code to work, and Flow will make sure it does work that way.

```
1 // @flow
2 function square(n: number): number {
3   return n * n;
4 }
5
6 square("2"); // Error!
```

Because Flow understands JavaScript so well, it doesn't need many of these types. You should only ever have to do a minimal amount of work to describe your code to Flow and it will *infer* the rest. A lot of the time, Flow can understand your code without any types at all.

```
1 // @flow
2 function square(n) {
3   return n * n; // Error!
4 }
5
6 square("2");
```

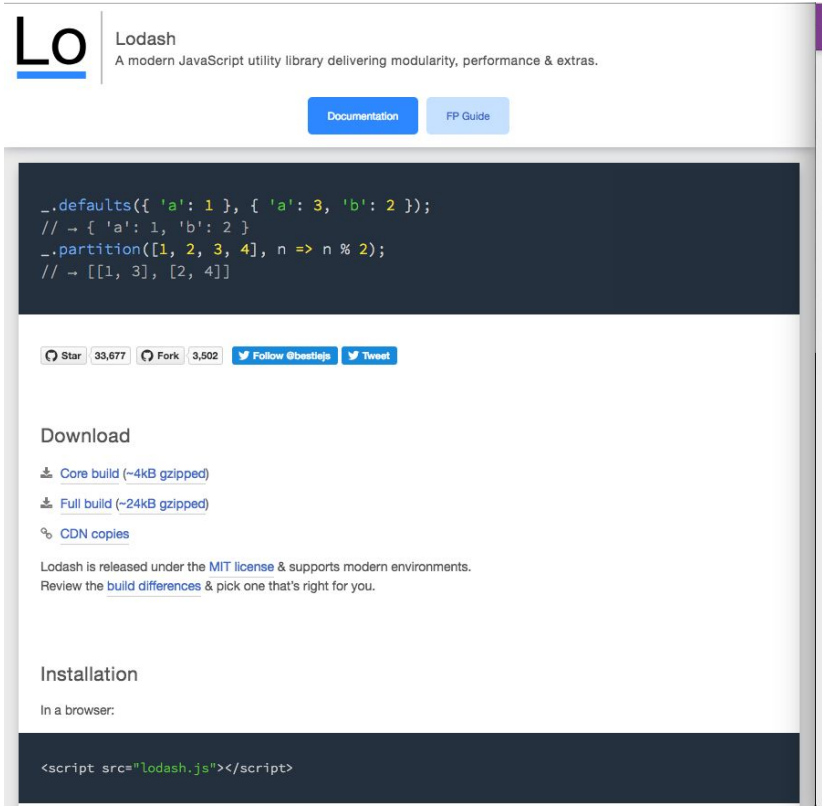
You can also adopt Flow incrementally and easily remove it at anytime, so you can try Flow out on any codebase and see how you like it.

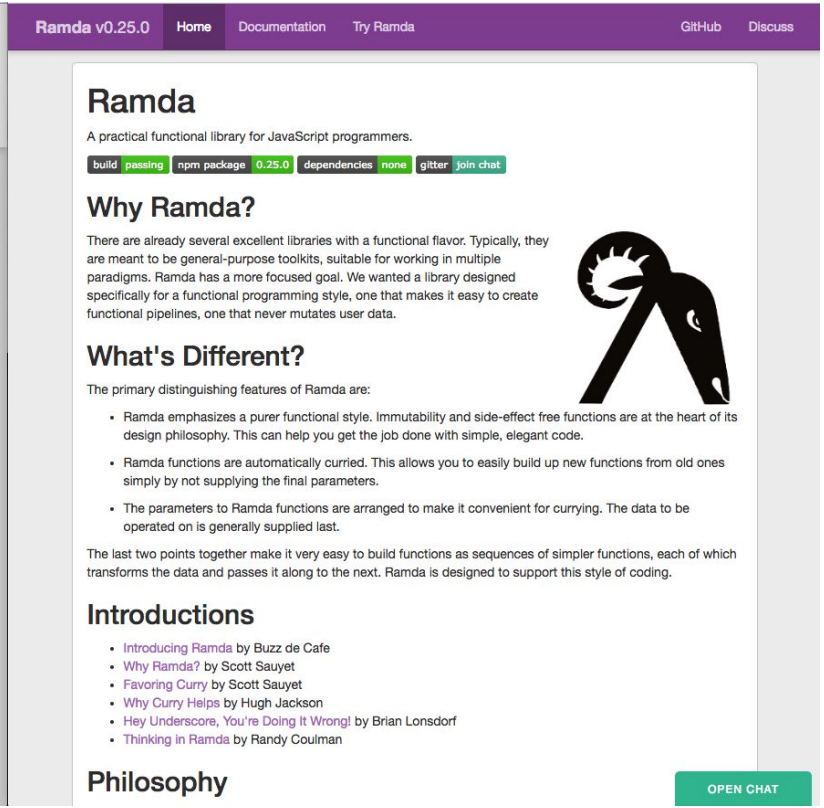
Installation →

> Was this guide helpful? Let us know by sending a message to **@flowtype**.


TypeScript
Quick Start
Documentation
Download
Connect
Playground
Fork me on GitHub
TypeScript
JavaScript that scales.
TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.
Any browser. Any host. Any OS. Open source.
Download
Documentation
#iHeartTypeScript
Dave Herman @littlecalculist
I ported my first nontrivial JS lib to @typescriptlang and it was a pure joy. What a lovely piece of technology.
Gabriela Mendes @Kappyh
@typescriptlang is really awesome O-O'
Thiago Script @thiagoviski
#TypeScript is really awesome! I'm glad to see people are using it in some of #Preact projects. #FrontEnd #Webpack #Programming

Libs


Lo
Lodash
A modern JavaScript utility library delivering modularity, performance & extras.
Documentation
FP Guide
_.defaults({ 'a': 1 }, { 'a': 3, 'b': 2 });
// → { 'a': 1, 'b': 2 }
_.partition([1, 2, 3, 4], n => n % 2);
// → [[1, 3], [2, 4]]
Star 33,677
Fork 3,502
Follow @bestiejs
Tweet
Download
Core build (~4kB gzipped)
Full build (~24kB gzipped)
CDN copies
Lodash is released under the MIT license & supports modern environments.
Review the build differences & pick one that's right for you.
Installation
In a browser:
<script src="lodash.js"></script>


Ramda v0.25.0
Home
Documentation
Try Ramda
GitHub
Discuss
Ramda
A practical functional library for JavaScript programmers.
build passing
npm package 0.25.0
dependencies none
gitter join chat
Why Ramda?
There are already several excellent libraries with a functional flavor. Typically, they are meant to be general-purpose toolkits, suitable for working in multiple paradigms. Ramda has a more focused goal. We wanted a library designed specifically for a functional programming style, one that makes it easy to create functional pipelines, one that never mutates user data.
What's Different?
The primary distinguishing features of Ramda are:
Ramda emphasizes a purer functional style. Immutability and side-effect free functions are at the heart of its design philosophy. This can help you get the job done with simple, elegant code.
Ramda functions are automatically curried. This allows you to easily build up new functions from old ones simply by not supplying the final parameters.
The parameters to Ramda functions are arranged to make it convenient for currying. The data to be operated on is generally supplied last.
The last two points together make it very easy to build functions as sequences of simpler functions, each of which transforms the data and passes it along to the next. Ramda is designed to support this style of coding.
Introductions
Introducing Ramda by Buzz de Cafe
Why Ramda? by Scott Sauyet
Favoring Curry by Scott Sauyet
Why Curry Helps by Hugh Jackson
Hey Underscore, You're Doing It Wrong! by Brian Lonsdorf
Thinking in Ramda by Randy Coulman
Philosophy
OPEN CHAT

Alguns conceitos funcionais

Funções como cidadãos de primeira classe

- Podem ser atribuídas a uma variável
- Podem retornar uma função
- Podem ser passadas como argumento para outras funções

```
function sum(a, b) {
   return a + b;
}
const subtract = function(a, b) {
   return a - b;
}
const operation = function(fn, a, b) {
    return fn(a, b);
};
operation(sum, 10, 3);
```

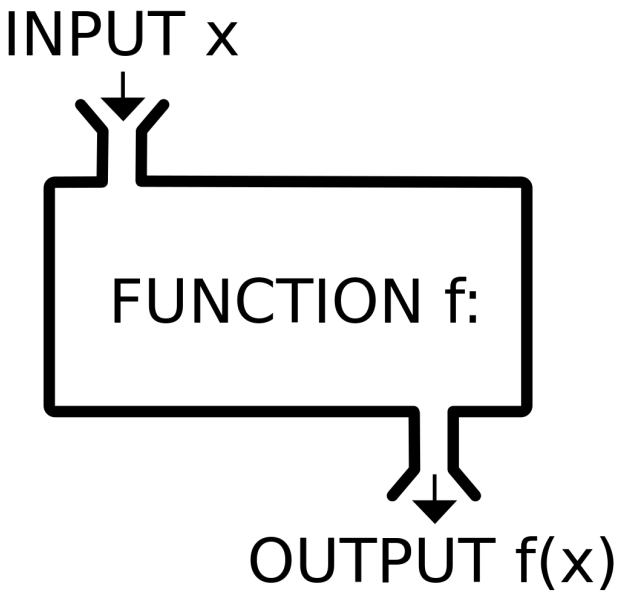
Funções puras

- Seu **valor de retorno** depende única e exclusivamente dos **parâmetros de entrada**
- É independente de qualquer estado externo.
- Sua execução não altera nada fora do próprio escopo, ou seja ela não gera **side-effects**

Side-effects

- Alteração de variáveis não-locais
- Chamada de funções que geram side-effects
- Interação com o DOM
- Escrita em arquivos, banco de dados
- Tudo o que interaja com o mundo externo e altere um estado não-local

Funções Puras




```
const double = function (x) {
    return 2 * x;
};
```

```
double x = 2 * x
```

```
const double = x => 2 * x;
```

Pura ou Impura?

Impura

```
let list = [];

const addItem = item => list.push(item);
```

```
const addItem = (list, item) => list.push(item);
```

Pura

```
const addItem = (list, item) => [...list, item];
```

Pura ou Impura?

Impura

```
const hasPermission = permission =>
    JSON.parse(
      localStorage.getItem('permissions')
     ).includes(permission);
```

Pura

```
const hasPermission = (permissions, permission) =>
   permissions.includes(permission);
```

Impura / side-effects

```
const updateVisitCount = () => {
  const current = parseInt(
    localStorage.getItem('total_visits') || 0, 10
  );
  localStorage.setItem('total_visits', current + 1);
}
```

Pura

```
const increment = v => v + 1;
```

```
const current = parseInt(localStorage.getItem('total_visits') || 0, 10);
localStorage.setItem(increment(current));
```

Imutabilidade, por que ela é importante?

Já se fez alguma dessas perguntas?

- Quando essa variável mudou de valor?
- Quem mudou essa variável?

Ou pior:

- Quando essa variável virou **`undefined`**?

Imutabilidade, por que ela é importante?

- As alterações em um programa deveriam seguir um fluxo contínuo no tempo em direção ao futuro. O passado existe apenas como read-only dentro de uma cadeia de eventos. Novos eventos são sempre adicionados no fim dessa linha do tempo.

- As modificações de estado do seu programa não devem modificar o estado original e sim retornar um novo estado.

- Imutabilidade está totalmente ligada a programação funcional, como quando trabalhamos funções puras, ajudando a evitar side-effects (Transparência referencial).

Imutabilidade no JavaScript

```
(1).toString(); // "1"
(1.31231232).toFixed(3); // "1.312"


"random value".toUpperCase(); // "RANDOM VALUE"
"carro".replace("rr", "r"); // "caro"
```

Imutabilidade no JavaScript

```
[1, 2, 3].concat([4, 5]); // [1, 2, 3, 4, 5]
[3, 4, 5].reverse(); // [3, 4, 5]


// mutable
[3, 4, 5].push(7); // [3, 4, 5, 7]
[1, 2, 3].pop(); // [1, 2]
[1, 2, 3].shift(); // [2, 3]
```

Imutabilidade no JavaScript

```
let a = "a";

let b = a;

b = b + "c";

console.log(a) // "a";

console.log(b) // "ac";
```

Imutabilidade no JavaScript

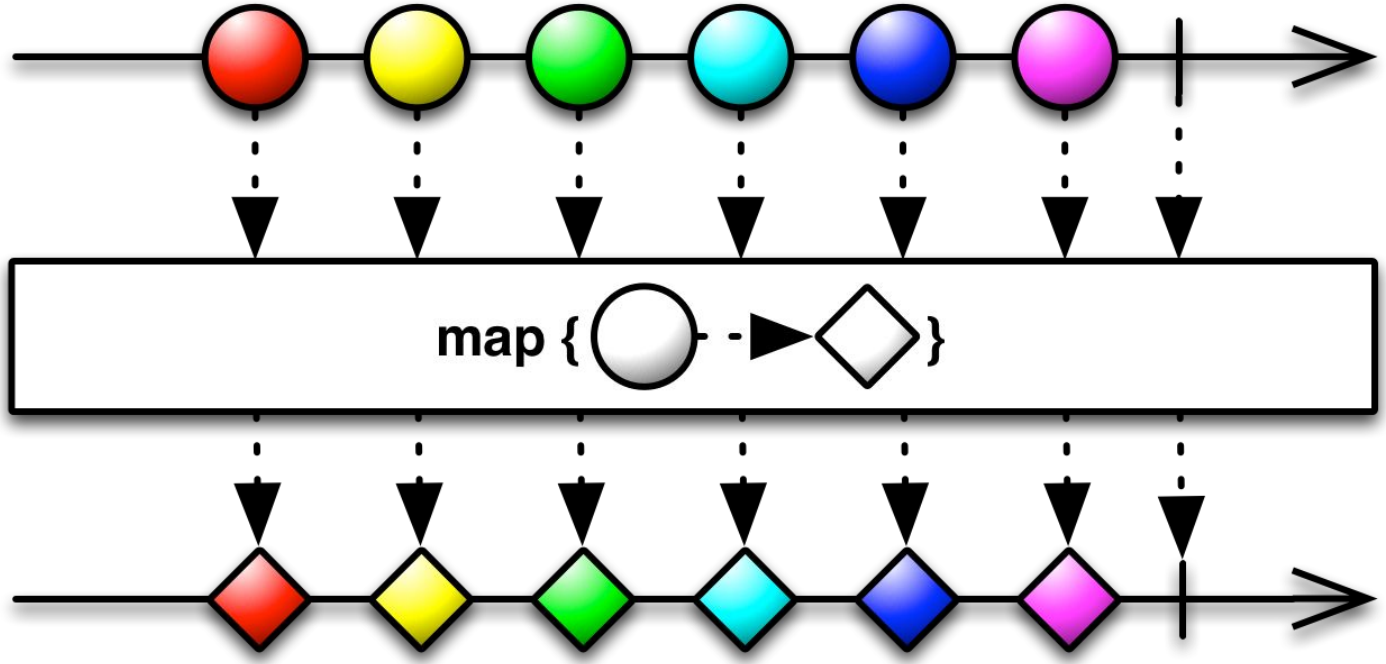
```
let person = { name: "Haskell" };

let person2 = person;

person2.name = "Haskell Curry";

console.log(person.name); // "Haskell Curry"
```

Imutabilidade no JavaScript

```
const numbers = [1, 2, 3];

const newNumbers = [...numbers, 4];

const person = { name: 'Haskell Curry' };

const newPerson = {...person, name: 'Alonzo Church'};


Object.freeze(person);

person.name = 'Leibniz'; // error in strict mode
```

O Básico de Funcional

map



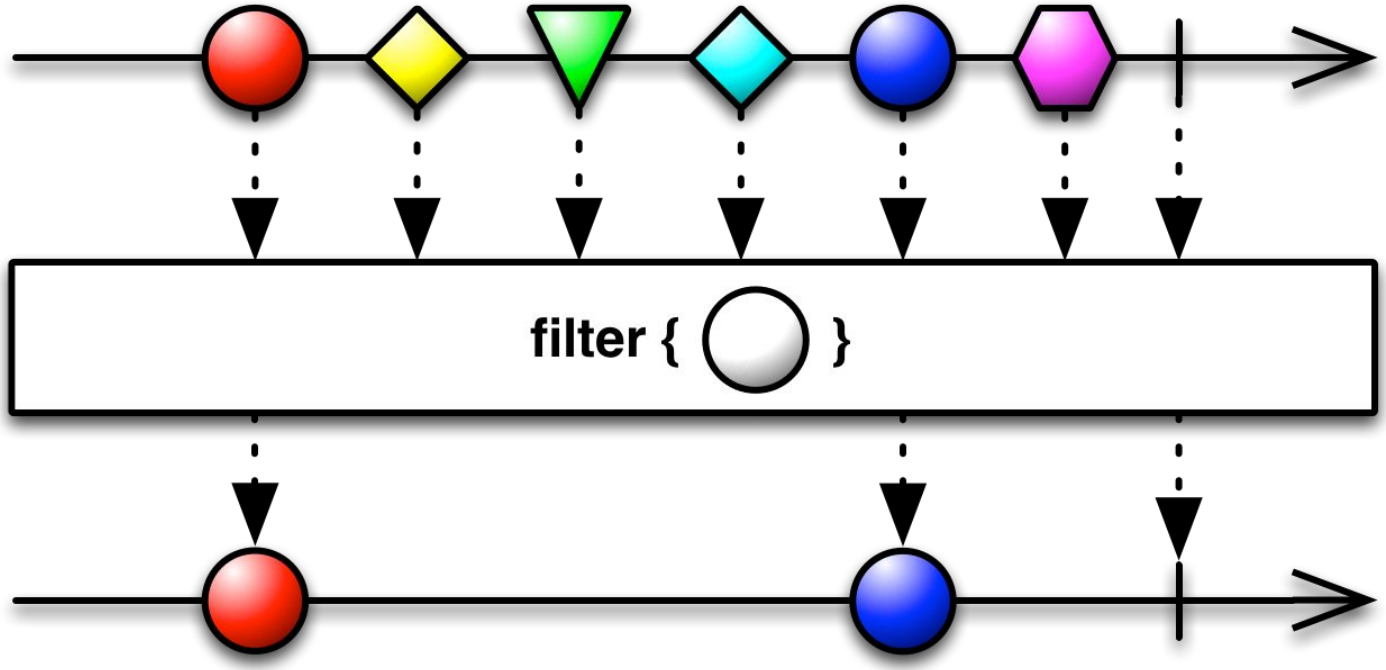
map

```
const users = [
   {id: 1, name: 'Haskell'},
   { id: 2, name: 'Church' }
];
let userNames = [];
for(let i=0; i < users.length; i++) {
   userNames.push(users[i].name);
}
console.log(userNames);
// ['Haskell', 'Church']
```

map

```
const users = [
   {id: 1, name: 'Haskell'},
   { id: 2, name: 'Church' }
];
const getName = user => user.name;
const userNames = users.map(getName);
console.log(userNames);
// ['Haskell', 'Church']
```

filter



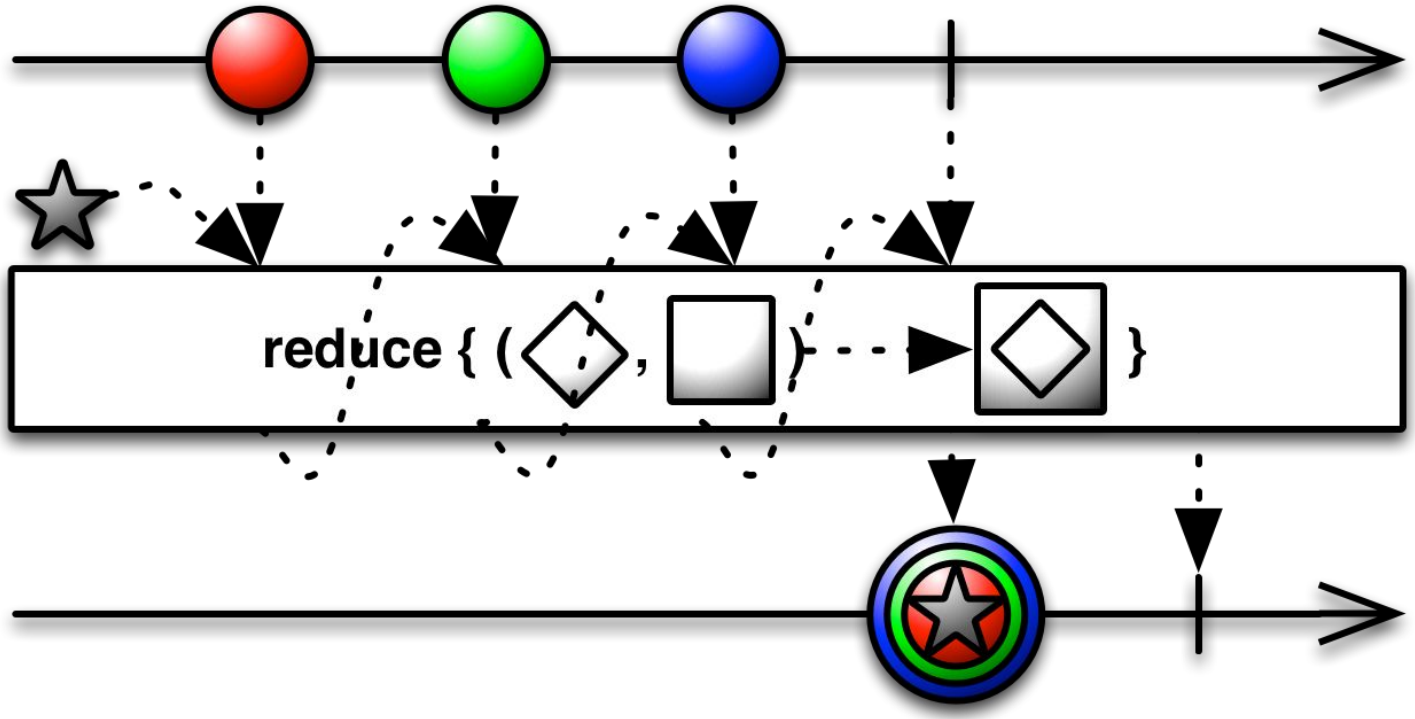
filter

```
let numbers = [20, 10, 13, 50, 5];
let result = [];
for(let i=0; i < numbers.length; i++) {
    if(numbers[i] > 10) {
        result.push(numbers[i]);
    }
}
console.log(result);
// [20, 13, 50]
```

filter

```
const numbers = [20, 10, 13, 50, 5];

const greaterThan10 = n => n > 10;

const result = numbers.filter(greaterThan10);

console.log(result);

// [20, 13, 50]
```

reduce



reduce

```
let values = [20, 40, 50, 100];

let result = 0;

for(let i=0; i < values.length; i++) {

   result = result + values[i];

}
```

reduce

```
const values = [20, 40, 50, 100];

const sum = (acc, value) => acc + value;

const result = values.reduce(sum);

console.log(result);

// 210
```

reduce

```
const products = [{ price: 10 }, { price: 22 }, { price: 12 }];
const calcTotal = (acc, value) => {
   acc.total += value.price;
   return acc;
};
const productsSum = products.reduce(calcTotal, { total: 0 });
console.log(productsSum);
// { total: 44 }
```

recursão

```
const recursiveSum = function (list) {
    const [first, ...rest] = list;
    if(list.length === 0) {
        return 0;
    }
    return first + recursiveSum(rest);
}
recursiveSum([10, 6, 20, 50]); // 86
```

curry

f(a, b, c)

f(a)

f(a, undefined, undefined)

f(a) -> f(b, c)

f(a, b) -> f(c)

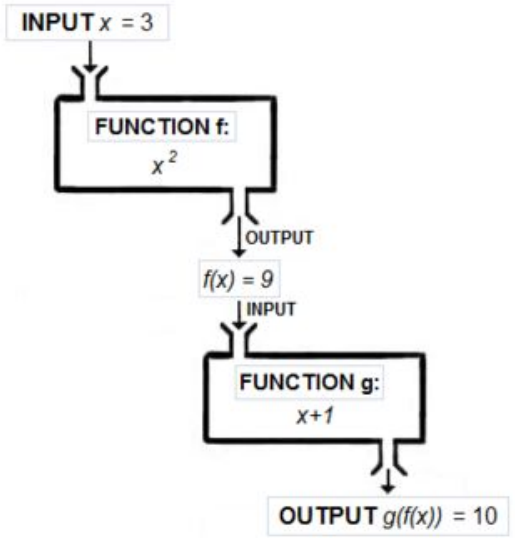
f(a, b, c) -> RESULTADO

curry

```
const sum = (a, b, c) => a + b + c;
const curriedSum = curry(sum);
curriedSum(1, 2, 3);
curriedSum(1)(2, 3);
curriedSum(1, 2)(3);

const multiply = (a, b) => a * b;
const double = curry(multiply)(2);
double(10) // 20;
```

Composição

Na **matemática**, uma **função composta** é criada aplicando uma função à saída, ou resultado, de uma outra função.



Composição

```
const increment = v => v + 1;

const double = v => 2  * v;

const decrement = v => v - 1;


increment(double(decrement(10))); // 19
```

Composição e Pipeline

```
const increment = v => v + 1;

const double = v => 2  * v;

const decrement = v => v - 1;


pipe(decrement, double, increment)(10);

compose(increment, double, decrement)(10); // 19
```

Composição e Pipeline

```
const add = (a, b) => a + b;
const increment = curry(add)(1);

const multiply = (a, b) => a * b;
const double = curry(multiply)(2);

pipe(double, increment)(10); // 21
compose(increment, double)(10); // 21
```

Pipeline Operator Proposal (|>)

```
const doubleSay = str => str + ", " + str;
const capitalize = str =>
    str[0].toUpperCase() + str.substring(1);
const exclaim = str => str + '!';
const result = "hello"
 |> doubleSay
 |> capitalize
 |> exclaim;
result // "Hello, hello!"
```

Obrigado!

Vagas para:

- Desenvolvedor **Front-End -** https://iclinic.workable.com/j/D3B6528552
- Desenvolvedor **Back-End-** https://iclinic.workable.com/jobs/647402