Programación Funcional

Introducción

- Programación Funcional
  - Eliminar el estado del programa
  - Expresar la computación con transformaciones de datos
  - Escribiendo y combinando funciones

Introducción

- Programación Funcional
  - Más fácil razonar sobre el código sin estado
  - Cada sección de código se puede entender aisladamente
  - Cada sección de código se puede **testear** aisladamente

Introducción

- Programación Funcional
  - Más fácil **predecir** el comportamiento del programa

Introducción

- Programación Funcional
  - Fomenta la **reutilización** del código

Introducción

- Programación Funcional
  - Mucho más fácil de **paralelizar**

Introducción

"Functional programming is programming without assignment statements."

Bob Martin

Introducción

- Aprender un nuevo lenguaje es “fácil”
  - memorizar detalles
- Aprender un nuevo paradigma es difícil
  - Cambiar la forma de pensar

Diccionario de Conceptos

estado

redradix

Diccionario de Conceptos

función pura

Diccionario de Conceptos

```
function suma(a, b) {
  return a + b;
}
```

Diccionario de Conceptos

```
function now() {
  return Date.now();
}
```

redradix

Diccionario de Conceptos

efecto secundario

Diccionario de Conceptos

```
let c = 0;
function counter() {
  return c++;
}
```

Diccionario de Conceptos

```
console.log('efecto secundario?');
```

Diccionario de Conceptos

programación declarativa

Diccionario de Conceptos

```
SELECT name, avatar FROM users;
```

Diccionario de Conceptos

```
$('ul.todo').find('.done').remove()
```

Diccionario de Conceptos

programación
imperativa

Diccionario de Conceptos

```
const numbers = [1, 2, 3, 4, 5];
let count = 0;
for (let i=0; i<numbers.length; i++) {
  if (numbers[i] % 2 === 0) {
    count++;
  }
}

console.log(count);
```

Diccionario de Conceptos

expresión

Diccionario de Conceptos

```
const suma = function(a, b) {
  return a + b;
};

const c = suma(2, 2 * 3);
```

Diccionario de Conceptos

```
const suma = function(a, b) {
  return a + b;
};

const c = suma(2, 2 * 3);
```

Diccionario de Conceptos

```
const suma = function(a, b) {
  return a + b;
};

const c = suma(2, 2 * 3);
```

redradix

Diccionario de Conceptos

sentencia

redradix

Diccionario de Conceptos

```
if (Math.random() > 0.5) {
  console.log('cara');
} else {
  console.log('cruz');
}
```

redradix

Diccionario de Conceptos

- FizzBuzz:
  - Escribe una función que devuelva los números del 1 al 100
  - Pero con los múltiplos de 3 sustituidos por la palabra “`fizz`”, los múltiplos de 5 por la palabra “`buzz`” y los múltiplos de 3 y 5 por la palabra “`fizzbuzz`”

Diccionario de Conceptos

```
[1, 2, "fizz", 4, "buzz", 6, ..., 14,
        "fizzbuzz", 16, ...]
```

redradix

```
function fizzbuzz() {
  const result = [];
  for (let i=1; i<=100; i++) {
    if ((i % 3 === 0) && (i % 5 === 0)) {
      result.push('fizzbuzz');
    } else if (i % 3 === 0) {
      result.push('fizz');
    } else if (i % 5 === 0) {
      result.push('buzz')
    } else {
      result.push(i);
    }
  }
  return result;
}
```

```
function range(start, end) {
  const list = [];
  for (let i=start; i<=end; i++)
    list.push(i);
  return list;
}
```

```
function mult3(n) { return n % 3 === 0; }
function mult5(n) { return n % 5 === 0; }

function and(pred1, pred2) {
  return n => pred1(n) && pred2(n);
}

function replaceWhen(pred, replacement) {
  return value => pred(value) ? replacement : value;
}
```

```
range(1, 100)
  .map(replaceWhen(and(mult3, mult5), 'fizzbuzz'))
  .map(replaceWhen(mult3, 'fizz'))
  .map(replaceWhen(mult5, 'buzz'));
```

Funciones de Orden Superior

Funciones de Orden Superior

- Funciones que operan sobre otras funciones
  - Recibiendo funciones como parámetros
  - Devolviendo funciones como valor de retorno
  - Nos permiten *abstraer acciones*

Guardad los ejercicios

Ejercicio: Unless

- Implementa **`unless(test, block)`**
  - utilidad de **control de flujo**
  - `ejecuta` **`block`** `cuando` **`test`** `es` **`false`**

Ejercicio: Unless

```
const env = 'DEBUG';

unless(env === 'PRODUCTION', () => {
  console.log('traza 18');
});
```

```
function unless(test, block) {
  if (!test) block();
}
```

Ejercicio: Repeat

- Implementa **`repeat(times, block)`**
  - utilidad de **control de flujo**
  - `ejecuta` **`times`** `veces` **`block`**

Ejercicio: Repeat

```
repeat(10, () => console.log('I <3 FP'));
```

```
function repeat(times, block) {
  for (let i = times; i--;) block();
}
```

Ejercicio: Once

- Implementa **`once(fn)`**
  - devuelve una función
  - que, al ejecutarse, invoca a **`fn`**
  - **pero sólo una vez**!

Ejercicio: Once

```
const log = once(console.log);

log('Hello!');
log('Goodbye!');
```

Ejercicio: Once

```
const log = once(console.log);

for (let i=0; i<100000; i++)
  log(i);
```

```
function once(fn) {
  let done = false;
  return (...args) => {
    if (done) return;
    done = true;
    return fn(...args);
  };
}
```

Ejercicio: Throttle

- Implementa **`throttle(fn, ms)`**
  - devuelve una función
  - que, al ejecutarse, ejecuta **fn**...
  - ...pero, cómo máximo, una vez cada **ms** milisegundos

Ejercicio: Throttle

```
const slowLog = throttle(console.log, 10);

slowLog('Hello!');
slowLog('Nop');
```

Ejercicio: Throttle

```
const slowLog = throttle(console.log, 10);

slowLog('Hello!');
setTimeout(() => slowLog(Bye!'), 11);
```

Ejercicio: Throttle

```
const slowLog = throttle(console.log, 10);

for (let i=0; i<100000; i++)
  slowLog(i);
```

```
function throttle(fn, ms) {
  let lastCall = 0;
  return (...args) => {
    let now = Date.now();
    if ((now - lastCall) > ms) {
      lastCall = now;
      return fn(...args);
    }
  }
}
```

Ejercicio: Debounce

- Implementa **`debounce(fn, ms)`**
  - devuelve una función
  - invoca a **`fn`** cuando hayan pasado **`ms`** milisegundos *tras la **última** llamada*.

Ejercicio: Debounce

```
const slowLog = debounce(console.log, 100);

slowLog('Hi in 100ms');
```

Ejercicio: Debounce

```
const slowLog = debounce(console.log, 100);

slowLog('Nop');
slowLog('Hi in 100ms');
```

Ejercicio: Debounce

```
const slowLog = debounce(console.log, 100);

slowLog('Nop');
setTimeout(() => slowLog('Hi in 110ms'), 10);
```

Ejercicio: Debounce

```
const slowLog = debounce(console.log, 100);

slowLog('Hi in 100ms');
setTimeout(() => slowLog('Hi in 201ms'), 101);
```

Ejercicio: Debounce

```
const slowLog = debounce(console.log, 10);

for (let i=0; i<100000; i++)
  slowLog(i);
```

```
function debounce(fn, ms) {
  let id = 0;
  return (...args) => {
    clearTimeout(id);
    id = setTimeout(() => fn(...args), ms);
  };
}
```

Ejercicio: Memoize

- Implementa **`memoize(fn)`**
  - **fn** tiene que ser una *función pura*
  - devuelve una función
  - cachea las llamadas a **fn**
  - la segunda vez que se llama a **fn** con un **mismo parámetro**, devuelve el resultado cacheado

Ejercicio: Memoize

```
function fib(n) {
  if (n === 0) return 0;
  if (n === 1) return 1;
  return fib(n - 1) + fib(n - 2);
}
```

Ejercicio: Memoize

```
const ffib = memoize(fib);

console.time('first time');
ffib(40);
console.timeEnd('first time');

console.time('second time');
ffib(40);
console.timeEnd('second time');
```

```
function memoize(fn) {
  const cache = {}
  return (value) => {
    if (cache[value] === undefined)
      cache[value] = fn(value)
    return cache[value]
  }
}
```

Ejercicio: Partial

- Implementa **`partial(fn, ...args)`**
  - “fija” un número de parámetros a **fn**
  - devuelve una función que recibe *menos* parámetros que **fn**
  - se explica mejor con un ejemplo

Ejercicio: Partial

```
const log = partial(console.log, 'She said:');

slowLog('Hello!'); // She said: Hello!

slogLog(); // She said:
```

Ejercicio: Partial

```
function suma(a, b) {
  return a + b;
}

const suma100 = partial(suma, 100);
suma100(2); // 102

const suma5y2 = partial(suma, 5, 2);
suma5y2(); // 7
```

```
function partial(fn, ...args) {
  return (...newargs) => fn(...args, ...newargs);
}
```

Ejercicio: Currify

- Implementa `currify(fn)`
  - aplicación parcial automática de **fn**
  - cada vez que se llama, devuelve una aplicación parcial
  - hasta que tiene **todos** los parámetros de **fn**

Ejercicio: Currify

```
function suma(a, b) { return a + b; }

const csuma = currify(suma);

csuma(1, 1); // 2

const suma1 = csuma(1);
suma1(1); // 2

csuma(1)(1); // 2
```

Ejercicio: Currify

```
function suma4(a, b, c, d) { return a + b + c + d; }

const rsuma4 = currify(suma4);

rsuma4(1, 1, 1, 1); // 4
rsuma4(1, 1, 1)(1); // 4
rsuma4(1, 1)(1, 1); // 4
rsuma4(1)(1, 1, 1); // 4
rsuma4(1)(1)(1)(1); // 4
```

```
function currify(fn) {
  return function aux(...args) {
    if (args.length >= fn.length)
      return fn(...args);
    else
      return (...more) => aux(...args, ...more);
  };
}
```

Operaciones Sobre Listas

Operaciones Sobre Listas

- Tres funciones fundamentales:
  - `map`
  - `filter`
  - `reduce`

Operaciones Sobre Listas

- `map(fn, list)`
  - devuelve un **nuevo array**
  - resultado de aplicar **`fn`** a cada elemento de **`list`**

Operaciones Sobre Listas

```
const suma = currify((a, b) => a + b);
const inicial = [1, 2, 3];
map(suma(100), inicial); // [101, 102, 103]
```

Ejercicio: Map

- Implementa `map(fn, list)`
  - devuelve un nuevo array
  - resultado de aplicar `fn` a cada elemento de `list`
  - **auto-currificada**

Ejercicio: Map

```
const suma = currify((a, b) => a + b);

const mapPlus100 = map(suma(100));
const mapPlus5 = map(suma(5));

mapPlus100([1, 2, 3]); // [101, 102, 103]
mapPlus5([1, 2, 3]); // [6, 7, 8]
```

```
const map = currify((fn, list) => {
  const result = [];
  for (let el of list)
    result.push(fn(el));
  return result;
});
```

Operaciones Sobre Listas

- `filter(fn, list)`
  - devuelve un **nuevo array**
  - sólo los elementos para los que **fn** es **true**

Operaciones Sobre Listas

```
const esPar = n => n % 2 === 0;
filter(esPar, [1, 2, 3, 4, 5]); // [2, 4]
```

Ejercicio: Filter

- Implementa `filter(fn, list)`
  - devuelve un nuevo array
  - sólo los elementos para los que `fn` es `true`
  - **auto-currificada**

```
const filter = currify((fn, list) => {
  const result = [];
  for (let el of list)
    if (fn(el)) result.push(el);
  return result;
});
```

Operaciones Sobre Listas

```
const pair = currify((a, b) => [a, b]);
const pack = fn => (args) => fn(...args);
const head = list => list[0];
const not = fn => (...args) => !fn(...args);
const gt = currify((a, b) => a > b);
```

Operaciones Sobre Listas

```
const listA = [1, 2, 3, 4, 5, 6];
const listB = [0, 0, 3, 2, 7, 1];


filter(gt(3), listA);
```

Operaciones Sobre Listas

```
const listA = [1, 2, 3, 4, 5, 6];
const listB = [0, 0, 3, 2, 7, 1];


filter(not(gt(3)), listB);
```

redradix

Operaciones Sobre Listas

```
const zip = currify((list1, list2) => {
  const len = Math.min(list1.length, list2.length);
  return map(i => pair(list1[i], list2[i]),
             range(0, len - 1));
})
```

```
zip([1, 2], [‘a’, ‘b’]) => [[1, ‘a’], [2, ‘b’]]
```

Operaciones Sobre Listas

```
const listA = [1, 2, 3, 4, 5, 6];
const listB = [0, 0, 3, 2, 7, 1];

map(head,
    filter(pack(gt), zip(listA, listB)));
```

Operaciones Sobre Listas

```
const misterio = currify((combine, start, list) => {
  let current = start;
  for (let element of list)
    current = combine(current, element);
  return current;
});
```

Operaciones Sobre Listas

- `reduce(combine, start, list)`
  - convierte una lista en otro valor cualquiera
  - aplicando secuencialmente una operación
  - a partir de un valor inicial dado

Operaciones Sobre Listas

```
const suma = (a, b) => a + b;
const sumaLista = reduce(suma, 0);

const list1 = [1, 1, 1, 1];
const list2 = [2, 2, 10];
sumaLista(list1); // 4
sumaLista(list2); // 14
```

Ejercicio: Reduce

- Implementa **`map`** y **`filter`** utilizando **`reduce`**

Operaciones Sobre Listas

```
function map2(fn, ...args) {
  const combine = (acc, el) => acc.concat(fn(el));
  return reduce(combine, [], ...args);
}
```

Operaciones Sobre Listas

```
function map2(fn, ...args) {
  const combine = (acc, el) => [...acc, fn(el)];
  return reduce(combine, [], ...args);
}
```

Operaciones Sobre Listas

```
function filter2(fn, ...args) {
  const combine = (acc, e) => fn(e) ? [...acc, e] : acc;
  return reduce(combine, [], ...args);
}
```

Operaciones Sobre Objetos

Operaciones Sobre Objetos

- Cuatro funciones interesantes:
  - `prop`
  - `assoc`
  - `mapKeys`
  - `mapValues`

Prop

```
const prop = currify((prop, obj) => obj[prop]);
```

Prop

```
const user = {name: 'antonio', pass: '1234'}
prop('name', user) // antonio

const getName = prop('name')
getName(user) // antonio


map(getName, users)
```

Assoc

```
const assoc = currify((prop, value, obj) => {
  obj[prop] = value;
  return obj;
});
```

Assoc

```
let keyValues = [['a', 1], ['b', 2]]

// Returns 2
keyValues.reduce((acc, kv) => acc[kv[ 0]] = kv[1], {})


// Returns {a: 1, b: 2}
keyValues.reduce((acc, kv) =>  assoc(kv[0], kv[1], acc), {})
```

Operaciones Sobre Objetos

- `mapKeys(fn, obj)`
  - construye **un nuevo objeto**
  - mapeando los nombres de las propiedades

Operaciones Sobre Objetos

```
const obj = { a: 1, b: 2 };
const toUpper = s => s.toUpperCase();
mapKeys(toUpper, obj); // { A: 1, B: 2 }
```

Ejercicio: mapKeys

- Implementa `mapKeys(fn, obj)`

Operaciones Sobre Objetos

```
const mapKeys = currify((fn, obj) => {
  const comb = (acc, key) => assoc(fn(key), obj[key], acc);
  return reduce(comb, {}, Object.keys(obj));
});
```

Operaciones Sobre Objetos

- `mapValues(fn, obj)`
  - construye **un nuevo objeto**
  - mapeando los valores de las propiedades

Operaciones Sobre Objetos

```
const obj = { a: 1, b: 2 };
const add10 = n => n + 10;
mapValues(add10, obj); // { a: 11, b: 12 }
```

Ejercicio: mapValues

- Implementa `mapValues(fn, obj)`

Operaciones Sobre Objetos

```
const mapValues = currify((fn, obj) => {
  const comb = (acc, el) => assoc(el, fn(obj[el]), acc);
  return reduce(comb, {}, Object.keys(obj));
});
```

Operaciones Sobre Objetos

¿Cómo serían **`filterKeys`** y **`filterValues`**?

Operaciones Sobre Objetos

```
const filterKeys = currify((fn, obj) => {
  const combine = (acc, el) => {
    return fn(el) ? assoc(el, obj[el], acc) : acc;
  };
  return reduce(combine, {}, Object.keys(obj));
});
```

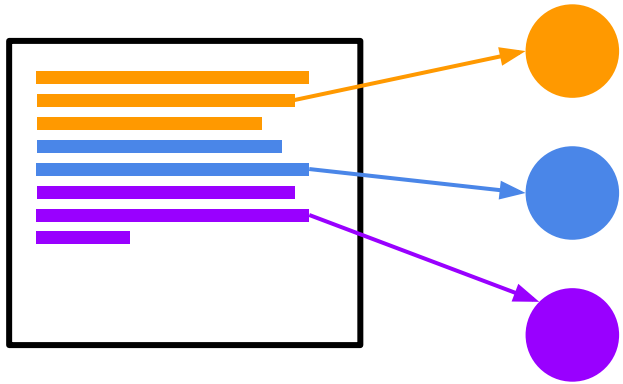
Composición de Funciones

redradix

Composición de Funciones

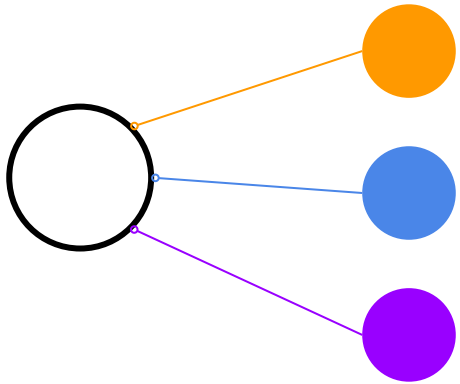
- Un *programa funcional* es semejante a un diccionario
    - Cada **función** define un nuevo concepto expresando una relación de elementos más simples
    - Para definir un **vocabulario de dominio** con el que se pueda expresar la solución

redradix

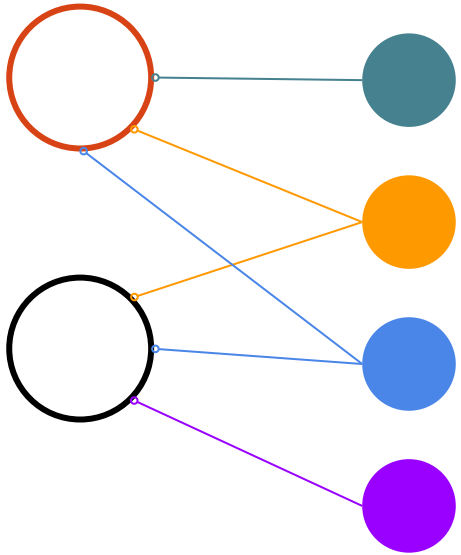


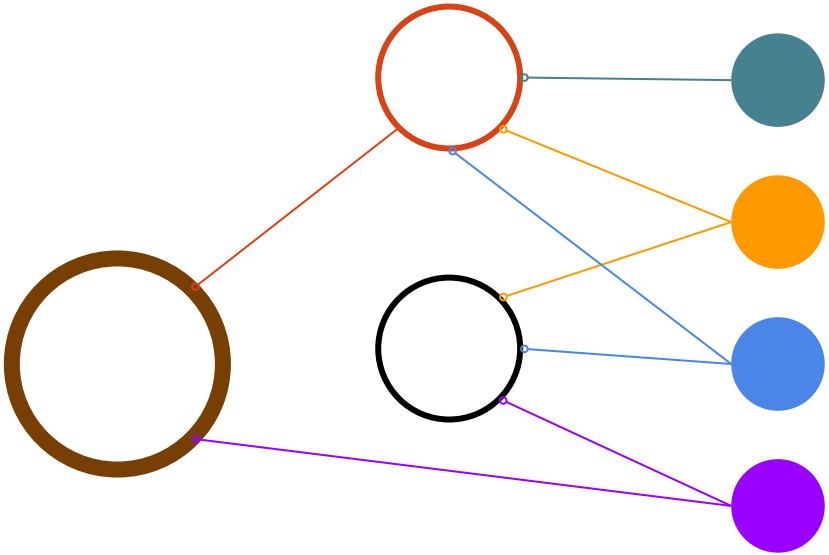
redradix

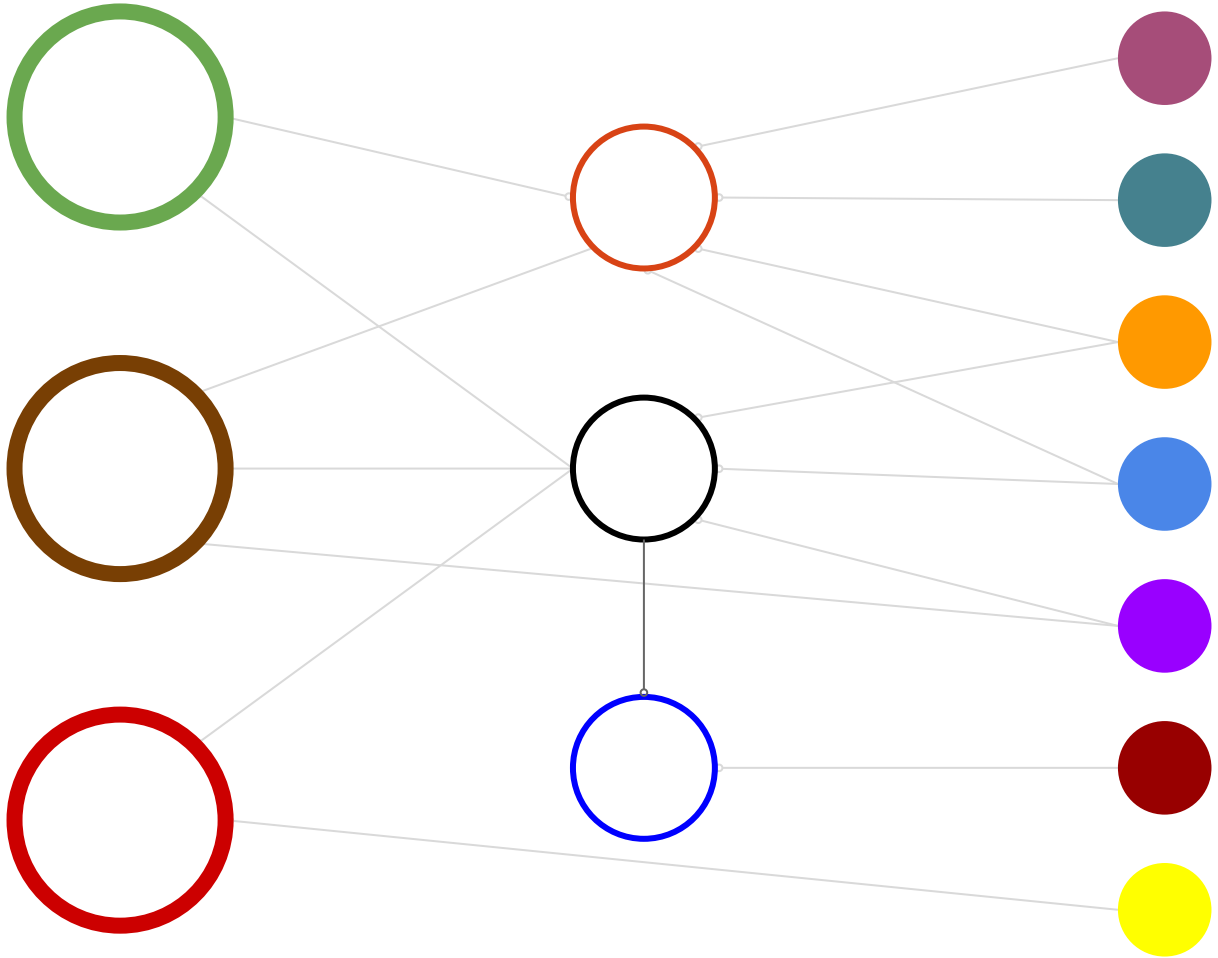
redradix



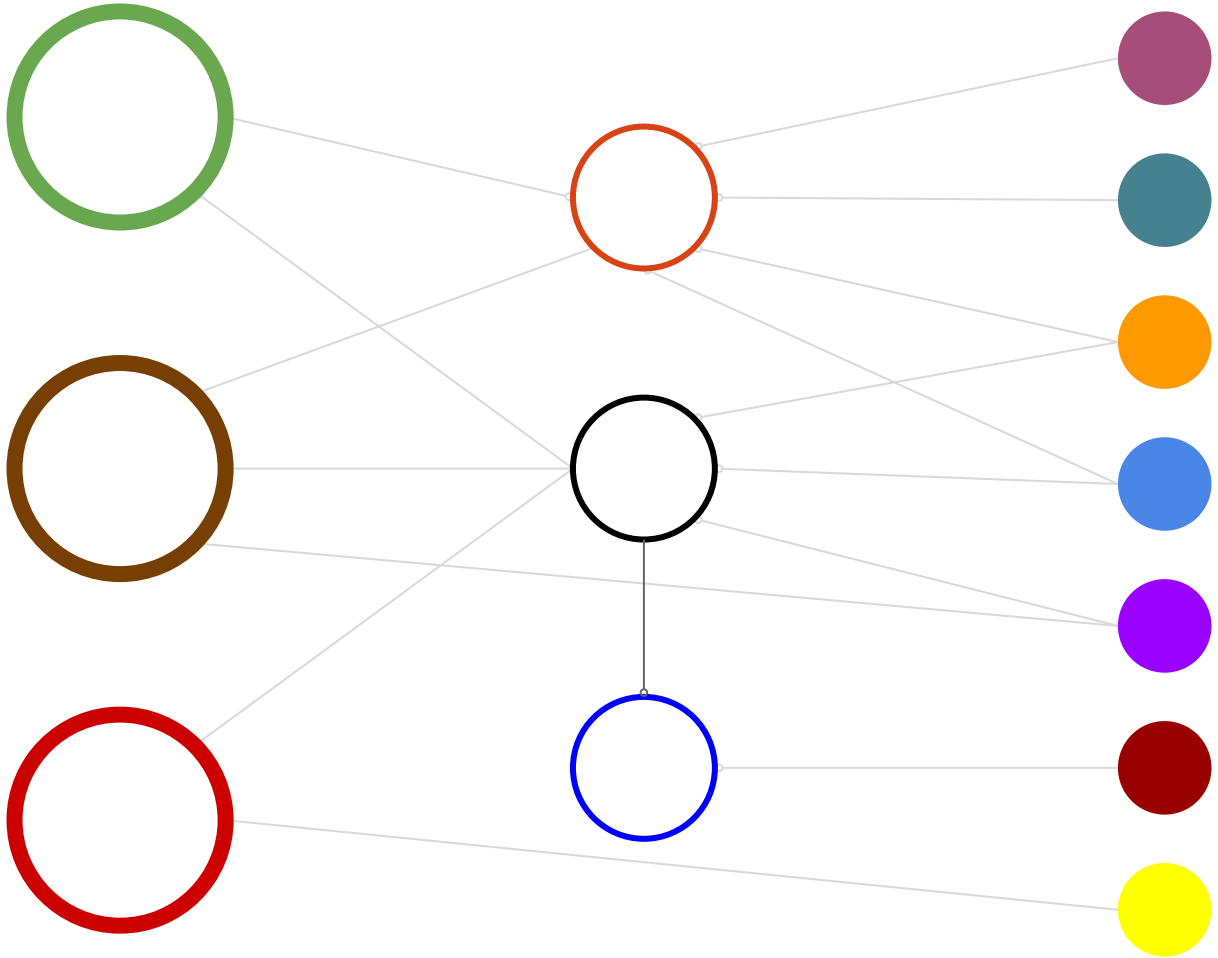
redradix







redradix



redradix

Composición de Funciones

- Un programa funcional se construye de *otra manera*
  - a partir de las herramientas que nos da el lenguaje
  - construímos muchas funciones muy sencillas
  - se **combinan** para **expresar** ideas más complejas
  - hasta construir un **vocabulario** lo bastante rico para expresar **qué queremos conseguir**
  - Estamos construyendo un lenguaje, no solucionando a base de algoritmia

Composición de Funciones

- La reutilización del código es muy alta
- El programa crece de manera natural
    - Cuanto más crece, más rico es su vocabulario
    - Cuanto más crece, más fácil es seguir extendiéndolo

Composición de Funciones

- La *programación funcional*, como paradigma, existe porque **las funciones tienen la propiedad de poder ser combinadas**.

Composición de Funciones

- Recursividad es la forma de composición más simple

Composición de Funciones

- `compose(fn1, fn2, fn3, …)`
  - Devuelve una **nueva función**
  - La operación de composición clásica
  - Crear nuevas operaciones a partir de operaciones existentes
  - `compose(a, b)(x) === a(b(x))`

Composición de Funciones

```
const suma = (a, b) => a + b;
const half = x => x / 2;

compose(half, suma)(10, 2) === half(suma(10, 2));
```

Ejercicio: compose

- Implementa **`compose(fn1, fn2, ...)`**
  - Todas las funciones reciben un solo parámetro
  - Excepto la **primera**, que puede recibir cualquier número de parámetros

Composición de Funciones

- **compose** es muy útil para crear **configuraciones** específicas de funcionalidad existentes
  - para **map**, **reduce**, **filter**, etc...
- Su utilidad depende del **catálogo de utilidades** que tengamos disponible para combinar

Composición de Funciones

```
const floor = Math.floor;
const random = Math.random;
const mul = currify((a, b) => a * b);
const exp = currify((a, b) => a ** b);
const toString = currify((b, n) => n.toString(b));

const rand10 = compose(floor, mul(10), random);
const rand53 = compose(floor, mul(53), random);
const randString = compose(
  toString(36), floor, mul(exp(36, 5)), random
);
```

Composición de Funciones

- `pipe(fn1, fn2, fn3, …)`
  - Igual que **compose**, pero con los parámetros en el orden inverso
  - **fn1** se ejecuta primero y el resultado se pasa a **fn2**, etc...
  - `pipe(a, b)(x) === b(a(x))`

Composición de Funciones

```
pipe(suma, half)(10, 2) === half(suma(10, 2));

const rand10 = pipe(rand, mul(10), floor);
```

Ejercicio: pipe

- Implementa **`pipe(fn1, fn2, ...)`**
  - Como una **composición** de otras funciones
  - `const pipe = compose(....)`
  - Escribe las **funciones auxiliares** que necesites

Composición de Funciones

```
const pipe = compose(pack(compose), unpack(reverse));
```

redradix

Composición de Funciones

```
const pack = fn => (args) => fn(...args);
const unpack = fn => (...args) => fn(args);
const reverse = list => list.reverse();

const pipe = compose(pack(compose), unpack(reverse));
```

Ejercicio: fizzbuzz

- Implementa **`fizzbuzz`**
  - Como una **composición** de otras funciones
  - `const fizzbuzz = compose(....)`
  - range(1, 100).map(fizzbuzz);
  - Escribe las **funciones auxiliares** que necesites

Composición de Funciones

```
const fizzbuzz = compose(
  replaceWhen(mult3, 'fizz'),
  replaceWhen(mult5, 'buzz'),
  replaceWhen(and(mult3, mult5), 'fizzbuzz')
);

range(1, 100).map(fizzbuzz);
```

Composición de Funciones

```
const fizzbuzz = map(compose(
  replaceWhen(mult3, 'fizz'),
  replaceWhen(mult5, 'buzz'),
  replaceWhen(and(mult3, mult5), 'fizzbuzz')
));

fizzbuzz(range(1, 100));
```

Composición de Funciones

```
const fizzbuzz = compose(
  map(compose(
    replaceWhen(mult3, 'fizz'),
    replaceWhen(mult5, 'buzz'),
    replaceWhen(and(mult3, mult5), 'fizzbuzz')
  )),
  range(1)
);

fizzbuzz(100);
```

Composición de Funciones

- `branch(testFn, trueFn, falseFn)`
  - Composición condicional
  - La versión funcional del condicional ternario

Composición de Funciones

```
const cara = partial(console.log, 'cara');
const cruz = partial(console.log, 'cruz');
const rand100 = compose(floor, mul(100), rand);
const condition = compose(gt(50), rand100);

const tirada = branch(condition, cara, cruz);
```

Ejercicio: branch

- Implementa **`branch(test, trueFn, falseFn)`**
  - devuelve una función
  - al ejecutarse, ejecuta **test**
    - si devuelve **true**, ejecuta **trueFn**
    - si deuvelve **false**, ejecuta **falseFn**

Composición de Funciones

```
const branch = (cond, tbranch, fbranch) => (...args) => {
  return cond(...args) ? tbranch(...args) : fbranch(...args);
}
```

Composición de Funciones

- `maybe(fn)`
  - Ejecución condicional para prevenir errores
  - Solo ejecuta **fn** si es invocada con un parámetro *truthy* o `0`

Composición de Funciones

```
const toString = n => n.toString();
toString(null); // throws!

const maybeToString = maybe(toString);
maybeToString(12); // -> "12";
maybeToString(null); // -> undefined (sin throw)
```

Composición de Funciones

- **compose**, **branch** y **maybe** son ejemplos de la filosofía funcional
  - pequeñas utilidades que expresan la relación entre diferentes elementos
  - aunque el código imperativo sea parecido, usando **compose/branch/maybe** la relación es explícita y clara