

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

Diccionario de Conceptos

función pura

Diccionario de Conceptos

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

Diccionario de Conceptos

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

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);
```


Diccionario de Conceptos

sentencia

Diccionario de Conceptos

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

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, ...]
```

```
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*

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 = new Map();  
  return (arg) => {  
    if (!cache.has(arg))  
      cache.set(arg, fn(arg));  
    return cache.get(arg);  
  };  
}
```

```
function memoize(fn) {  
  const cache = new Map();  
  return (...args) => {  
    const key = args;  
    if (!cache.has(key))  
      cache.set(key, fn(...args));  
    return cache.get(key);  
  };  
}
```

```
function memoize(fn) {  
  const cache = new Map();  
  return (...args) => {  
    const key = JSON.stringify(args);  
    if (!cache.has(key))  
      cache.set(key, fn(...args));  
    return cache.get(key);  
  };  
}
```

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: Partial Right

- Implementa **partialRight(fn, ...args)**
 - aplicación parcial de **fn**
 - por la derecha

Ejercicio: Partial Right

```
function operacion(a, b, c) {  
  return a * b - c;  
}
```

```
const op1 = partialRight(operacion, 1, 3);  
op1(2); // 5
```

```
const log = console.log;  
partialRight(log, 'cuatro', 'tres')('uno', 'dos');
```

```
function reverse(list) {  
  const newList = [];  
  for (let i=list.length; i--;)  
    newList.push(list[i]);  
  return newList;  
}
```

```
function partialRight(fn, ...args) {  
  const rargs = reverse(args);  
  return (...newargs) => fn(...newargs, ...rargs);  
}
```

```
function partialRight(fn, ...args) {  
  const rargs = args.reverse();  
  return (...newargs) => fn(...newargs, ...rargs);  
}
```

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: Curryfy

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

```
const rsuma = currfy(suma);
```

```
rsuma(1, 1); // 2
```

```
const suma1 = rsuma(1);
```

```
suma1(1); // 2
```

```
rsuma(1)(1); // 2
```


Ejercicio: Curryfy

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

```
const rsuma4 = curify(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 currfy(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 = curify((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 = curify((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 = currfy((fn, list) => {  
  const result = [];  
  for (let el of list)  
    result.push(fn(el));  
  return result;  
});
```

Ejercicio: Each

- Implementa **each(fn, list)**
 - aplica la función a cada elemento de la lista
 - no devuelve nada
 - sólo para efectos secundarios
 - no es funcional!

```
const log = console.log;  
const logEach = each(log);  
logEach(['a', 'b', 'c']);
```

```
const logFizzBuzz = each((i) => {  
  const fizz = i % 3 === 0;  
  const buzz = i % 5 === 0;  
  if (fizz && buzz) log('fizzbuzz');  
  else if (fizz) log('fizz');  
  else if (buzz) log('buzz')  
  else log(i);  
});
```

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

```
const each = curify((fn, list) => {  
  for (let el of list) fn(el);  
});
```

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


Operaciones Sobre Listas

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

Operaciones Sobre Listas

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

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);
```

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 = curify((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

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

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`

Operaciones Sobre Objetos

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

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


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

Operaciones Sobre Objetos

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

Operaciones Sobre Objetos

```
const obj = { a: 1, b: 2 };
```

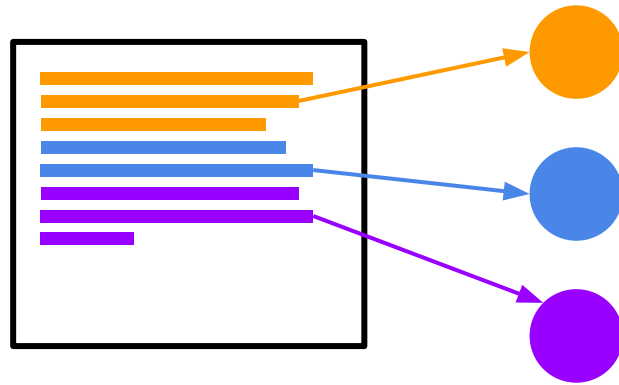
```
map(([v, k]) => `${v} => ${k}`,  
    Object.entries(obj));
```

Composición de Funciones

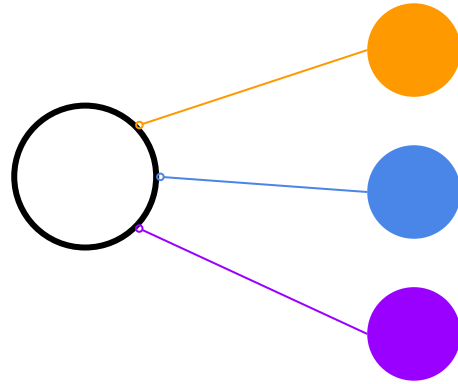
Composición de Funciones

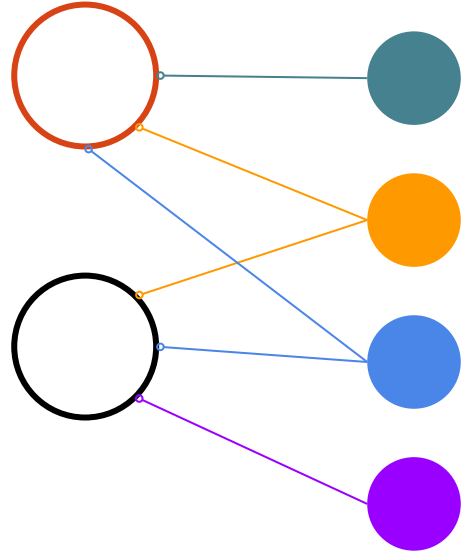
- Un programa funcional es semejante a un diccionario
 - Cada **función** define un nuevo concepto expresando una relación de elementos más simples
 - Para definir un **vocabulario de dominio** con el que se pueda expresar la solución

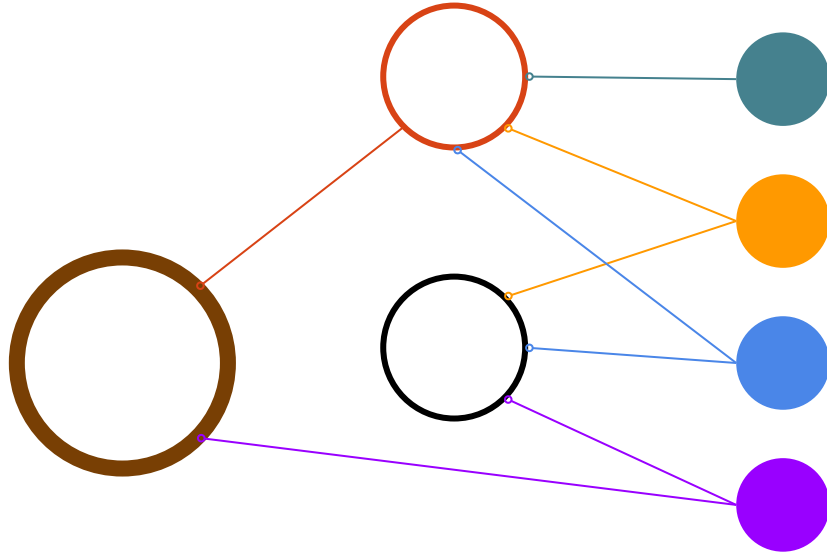


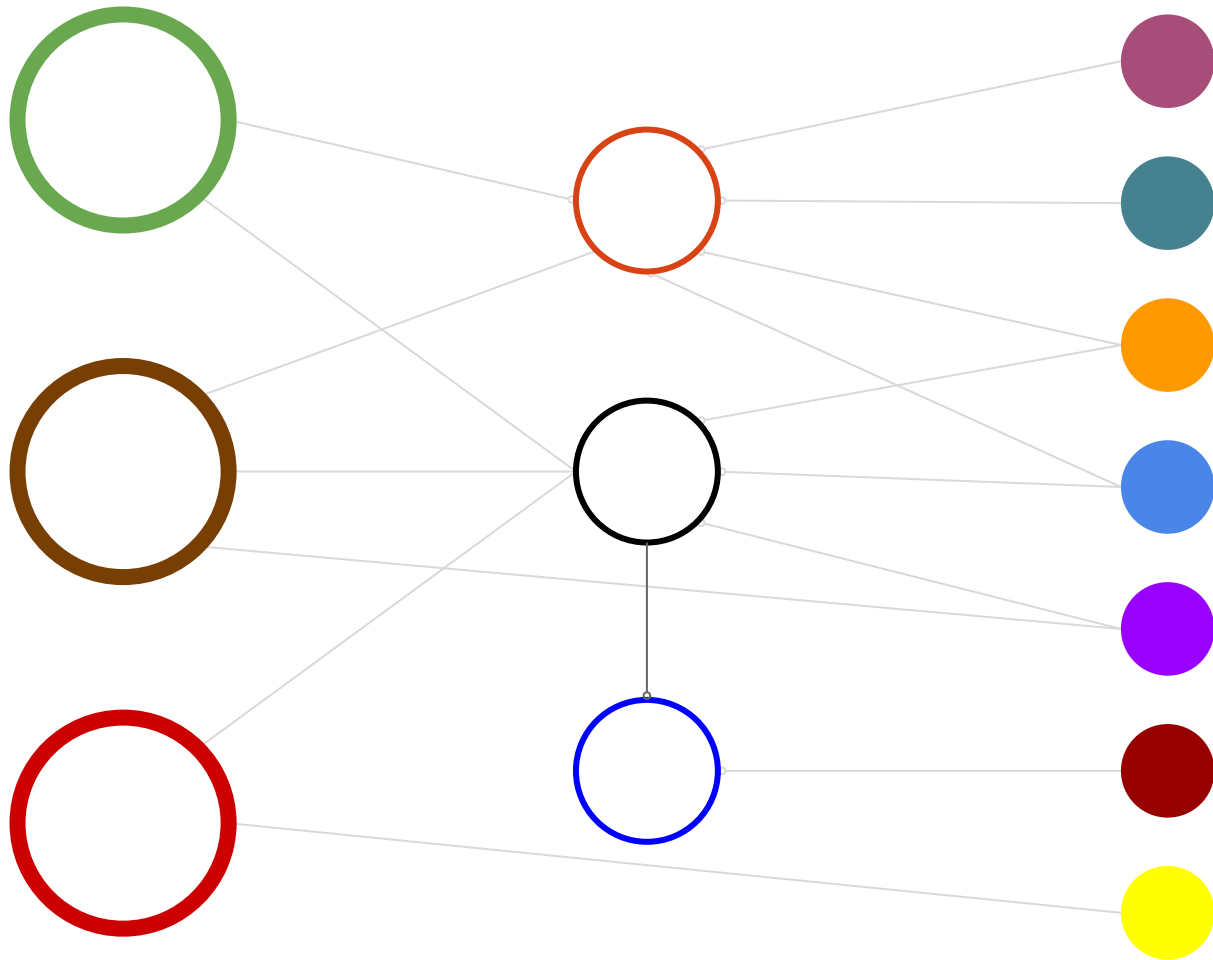


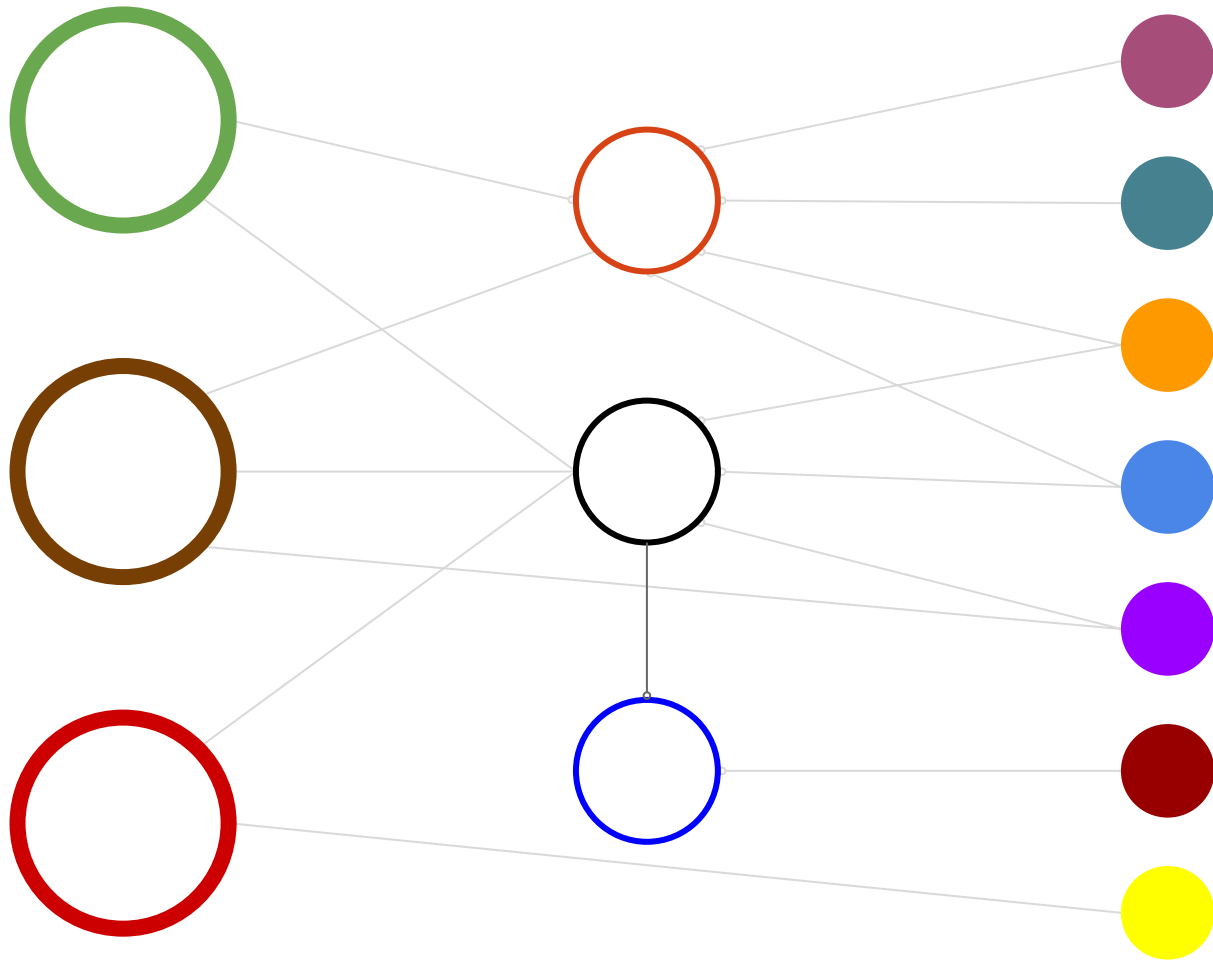












Composición de Funciones

- Un programa funcional se construye de *otra manera*
 - a partir de las herramientas que nos da el lenguaje
 - construimos muchas funciones muy sencillas
 - se **combinan** para **expresar** ideas más complejas
 - hasta construir un **vocabulario** lo bastante rico para expresar **qué queremos conseguir**
 - pensamos en **resultados**, no en pasos

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

- Las funciones *quieren* ser combinadas!
 - **factorizadas** en funciones más simples
 - ofrecer una **abstracción bien definida que pueda ser combinada** para expresar otra función de nivel superior

Composición de Funciones

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

Composición de Funciones

```
const head = list => list[0];  
const tail = list => list.slice(1);  
  
const filter = curify((fn, list) => {  
  if (list.length === 0) return list;  
  if (fn(head(list))) return [head(list), ...filter(fn, tail(list))];  
  return filter(fn, tail(list));  
});
```

Composición de Funciones

- Recursividad es la composición más simple
- Recursividad > iteración
 - más expresiva
 - **quién gestiona el estado?**

Ejercicio: **filterTree**

- Implementa **filterTree(fn, tree)**
 - **tree**: listas anidadas
 - elimina las **hojas** que para las que **fn** devuelve **false**

Ejercicio: filterTree

```
const odd = n => n % 2 === 1;  
  
const tree = [[1, [2, 3]], 4, [5, [6, [7]]]];  
  
console.log(filterTree(odd, tree));  
// [[1,[3]], [5,[[7]]]]
```

Ejercicio: filterTree

- Escribe una implementación **recursiva**
- Y otra implementación **iterativa**

Funcional vs. Imperativo

- Un programa funcional empieza *en el algoritmo*
 - un algoritmo imperativo \Rightarrow código imperativo
 - algoritmo funcional \Rightarrow código funcional

Ejercicio: recursividad

- Implementa `sumUntil(n)`
 - Devuelve la suma de todos los números naturales
 - desde 0
 - hasta `n`
 - **No** utilices bucles ni variables

Ejercicio: recursividad

- Implementa **strCount(haystack, needle)**
 - **haystack** y **needle** son strings
 - Cuenta cuantas veces **needle** aparece en **haystack**
 - **No** utilices bucles ni variables

Composición de Funciones

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

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 = curify((a, b) => a * b);
const exp = curify((a, b) => a ** b);
const toString = curify((b, n) => n.toString(b));

const rand10 = compose(floor, mul(100), 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));
```

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(...)`
 - 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 devuelve **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

- `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)
```


Ejercicio: branch

- Implementa **maybe(fn)**
 - implementa las utilidades que necesites

Composición de Funciones

```
const maybe = fn => (v, ...args) => {  
  if (!(v === null || v === undefined || v === NaN)) {  
    return fn(v, ...args);  
  }  
};
```

Composición de Funciones

```
const maybe = fn => (v, ...args) => {  
  if (!(v === null || v === undefined || Number.isNaN(v))) {  
    return fn(v, ...args);  
  }  
};
```

Composición de Funciones

```
const maybe = fn => when(isNotNil, fn);
```

Composición de Funciones

```
const constantly = v => () => v;  
const when = (cond, fn) => branch(cond, fn, constantly(undefined));  
  
const isNotNil = v => !(  
  v === null || v === undefined || Number.isNaN(v)  
);  
const maybe = fn => when(isNotNil, fn);
```

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

Composición de Funciones

```
const maybe = fn => (v, ...args) => {  
  if (!(v === null || v === undefined || Number.isNaN(v))) {  
    return fn(v, ...args);  
  }  
};
```

. vs .

```
const maybe = fn => when(isNotNil, fn);
```

Programación Tácita

Programación Tácita

- Point-free programming
- Definir funciones componiendo otras funciones
- Sin hacer explícitos sus argumentos
- Point-free = sin argumentos

Programación Tácita

```
function withPoint(x) {  
  const y = foo(x);  
  const z = bar(y);  
  const w = baz(z);  
  return w;  
}
```

Programación Tácita

```
const pointFree = compose(baz, bar, foo);
```

Programación Tácita

- Máxima reutilización de código!
- Ayuda a **pensar en componer funciones** (*alto nivel*)...
- ...en vez de pensar en manipular data (*bajo nivel*)
- Fomenta la creación de un **vocabulario rico y ortogonal** del dominio

Programación Tácita

- No es un paradigma
- Es un **estilo ideal** al que deberíamos tender
- El “nirvana” de la programación funcional
- Síntoma de un programa **maduro y bien diseñado**
- Característico de las capas “altas” del programa

Ejercicio: **fizzbuzz**

- Implementa **fizzbuzz**
 - utilizando **branch**
 - `const fizzbuzz = branch(...)`
 - Escribe las **funciones auxiliares** que necesites
 - intenta crear utilidades que se combinen bien
 - para crear un código legible y claro

Composición de Funciones

```
const fizzBuzzNumber = branch(  
  isFizzBuzz,  
  constantly('fizzbuzz'),  
  branch(  
    isFizz,  
    constantly('fizz'),  
    branch(isBuzz, constantly('buzz'), identity)  
  )  
);
```

```
const fizzBuzz = partial(map, fizzBuzzNumber,  
  range(100));
```

Composición de Funciones

```
const constantly = v => () => v;
```

```
const identity = v => v;
```

```
const fevery = (...fns) => (...args) => {  
  return reduce((acc, fn) => acc && fn(...args), true, fns);  
}
```

```
const isDivisibleBy = n => v => v % n === 0;
```

```
const isFizz = isDivisibleBy(3);
```

```
const isBuzz = isDivisibleBy(5);
```

```
const isFizzBuzz = fevery(isFizz, isBuzz);
```


Programación Tácita

- Al escribir código funcional...
 - es muy importante **estar alerta**
 - para detectar **cuándo necesitamos** escribir
 - **una nueva utilidad**
 - que nos permite **expresarnos con más claridad**

Ejercicio: **fizzbuzz**

- Vuelva a implementar **fizzbuzz**
 - utilizando **una nueva utilidad**
 - que exprese la lógica con más claridad que **branch**
 - piensa: *“Cuál sería la mejor manera de expresar este algoritmo?”*

Composición de Funciones

```
const fizzBuzzNumber = cond(  
  [isFizzBuzz, constantly('fizzbuzz')],  
  [isFizz, constantly('fizz')],  
  [isBuzz, constantly('buzz')],  
  [constantly(true), identity]  
);
```

```
const fizzBuzz = partial(map, fizzBuzzNumber, range(100));
```

Composición de Funciones

```
const fizzBuzzNumber = cond(  
  [fevery(isDivisibleBy(3), isDivisibleBy(5)), constantly('fizzbuzz')],  
  [isDivisibleBy(3), constantly('fizz')],  
  [isDivisibleBy(5), constantly('buzz')],  
  [constantly(true), identity]  
);
```

```
const fizzBuzz = partial(map, fizzBuzzNumber, range(100));
```

Composición de Funciones

```
const head = list => list[0];  
const find = (fn, list) => list.find(fn);  
  
const cond = (...pairs) => (...args) => {  
  const [, action] = find(pair => head(pair)(...args), pairs);  
  return action(...args);  
}
```