Functional Programming

Introduction

- Functional Programming
  - Remove the program state
  - Express the computation as data transformation
  - Writing and combining functions

Introduction

- Functional Programming
  - Easier to reason about stateless code
  - Each section can be understood in isolation
  - Each section can be **tested** in isolation

Introduction

- Functional Programming
  - Easier to **predict** the behavior of stateless code

Introduction

- Functional Programming
  - Encourages **code reusability**

Introduction

- Functional Programming
  - Easier to **parallelize**

Introduction

"Functional programming is programming without assignment statements."

Bob Martin

Introduction

- Learn a new language is “easy”
  - memorize syntax
- Learn a new paradigm is difficult
  - change the way we think

Concepts

state

Concepts

pure function

Concepts

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

Concepts

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

Concepts

side effect

Concepts

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

Concepts

```
console.log('side effect?');
```

Concepts

declarative
programming

Concepts

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

Concepts

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

Concepts

imperative programming

Concepts

```
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);
```

Concepts

expression

Concepts

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

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

Concepts

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

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

Concepts

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

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

redradix

Concepts

statement

Concepts

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

Concepts

- FizzBuzz:
  - Write a function that returns the numbers from 1 to 100
  - But with multiples of 3 replaced by the word “`fizz`”, multiples of 5 by the word “`buzz`” and multiples of both 3 and 5 by the word “`fizzbuzz`”

Concepts

```
[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'));
```

Higher Order Functions

Higher Order Functions

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

Exercise: Unless

- `unless(test, block)`
  - **control flow** utility
  - runs **`block`** when **`test`** is **`false`**

Exercise: Unless

```
const env = 'DEBUG';

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

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

Exercise: Repeat

- `repeat(times, block)`
  - **control flow** utility
  - runs **`block`** **`times`** times

Exercise: Repeat

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

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

Exercise: Once

- `once(fn)`
  - returns a function
  - when run, invokes `fn`
  - **but only once**!

Exercise: Once

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

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

Exercise: 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);
  };
}
```

Exercise: Throttle

- `throttle(fn, ms)`
  - returns a function
  - that invokes **fn**...
  - ...but only once every **ms** milliseconds

Exercise: Throttle

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

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

Exercise: Throttle

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

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

Exercise: 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);
    }
  }
}
```

Exercise: Debounce

- **`debounce(fn, ms)`**
  - returns a function
  - invokes **`fn`** after **`ms`** milliseconds have passed *since the **last** call*.

Exercise: Debounce

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

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

Exercise: Debounce

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

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

Exercise: Debounce

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

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

Exercise: Debounce

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

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

Exercise: 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);
  };
}
```

Exercise: Memoize

- `memoize(fn)`
  - **fn** must be a *pure function*
  - returns a function
  - remembers the result of every call to **fn**
  - if called with the same parameters twice, it will **not** run **fn** again: it will just return the previous result.

Exercise: Memoize

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

Exercise: 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);
  };
}
```

Exercise: Partial

- `partial(fn, ...args)`
  - “fixes” a number of parameters to **fn**
  - returns a function that receives *less* parameters than **fn**

Exercise: Partial

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

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

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

Exercise: Partial

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

const add100 = partial(add, 100);
add100(2); // 102

const add5and2 = partial(add, 5, 2);
add5and2(); // 7
```

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

Exercise: Partial Right

- `partialRight(fn, ...args)`
  - partial application of **fn**
  - starting from the right

Exercise: Partial Right

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

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

const log = console.log;
partialRight(log, 'four', 'three')('one', 'two');
```

```
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);
}
```

Exercise: Currify

- `currify(fn)`
  - automatic partial application of **fn**
  - returns a function that, each time it is called, returns a partially applied version of itself
  - until it has **enough** parameters to call **fn**

Exercise: Currify

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

const radd = currify(add);

radd(1, 1); // 2

const add1 = radd(1);
add(1); // 2

radd(1)(1); // 2
```

Exercise: Currify

```
function add4(a, b, c, d) { return a + b + c + d; }

const radd4 = currify(add4);

radd4(1, 1, 1, 1); // 4
radd4(1, 1, 1)(1); // 4
radd4(1, 1)(1, 1); // 4
radd4(1)(1, 1, 1); // 4
radd4(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);
  };
}
```

List Operations

List Operations

- Three fundamental operations:
  - `map`
  - `filter`
  - `reduce`

List Operations

- `map(fn, list)`
  - returns a **new array**
  - with the results of applying **fn** to every element of **list**

List Operations

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

Exercise: Map

- `map(fn, list)`
  - returns a new array
  - with the results of applying fn to every element of list
  - **curryfied!**

Exercise: Map

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

const mapPlus100 = map(add(100));
const mapPlus5 = map(add(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;
});
```

Exercise: Each

- `each(fn, list)`
  - applies **fn** to every element of **list**
  - doesn't have a return value
    - just for side effects
    - not functional!

```
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));
```

redradix

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

List Operations

- `filter(fn, list)`
  - returns a **new array**
  - only with the elements of **list** for which **fn** returned true

List Operations

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

Exercise: Filter

- `filter(fn, list)`
  - returns a new **array**
  - only with the elements of **list** for which **fn** is **true**
  - **curryfied!**

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

List Operations

```
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);
```

List Operations

```
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));
})
```

List Operations

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


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

List Operations

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


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

List Operations

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

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

List Operations

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

List Operations

- `reduce(combine, start, list)`
  - transforms **list** into any other value
  - applying **combine** sequentially
  - starting with a given value **start**

List Operations

```
const add = (a, b) => a + b;
const addList = reduce(add, 0);

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

Exercise: Reduce

- implement **`map`** and **`filter`** using **`reduce`**

List Operations

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

List Operations

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

List Operations

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

List Operations

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

Object Operations

Object Operations

- Four interesting operations:
  - `prop`
  - `assoc`
  - `mapKeys`
  - `mapValues`

Object Operations

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

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

Object Operations

- `mapKeys(fn, obj)`
  - returns a **new object**
  - mapping the *property names*

Object Operations

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

Exercise: mapKeys

- Implement `mapKeys(fn, obj)`

Object Operations

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

Object Operations

- `mapValues(fn, obj)`
  - returns a **new object**
  - mapping the *property values*

Object Operations

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

Exercise: mapValues

- Implement `mapValues(fn, obj)`

Object Operations

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

Object Operations

What **`filterKeys`** and **`filterValues`** would be like?

Object Operations

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

Object Operations

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

Object Operations

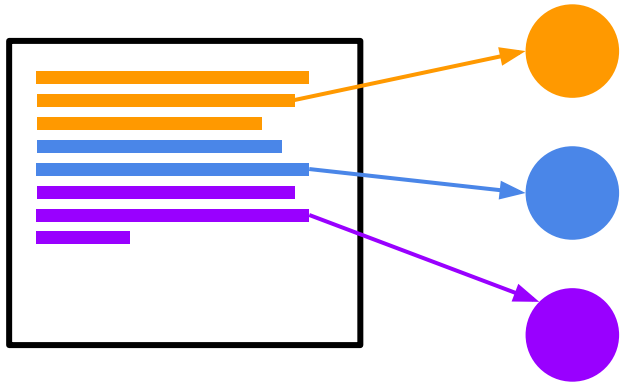
```
const obj = { a: 1, b: 2 };

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

Function Composition

Function Composition

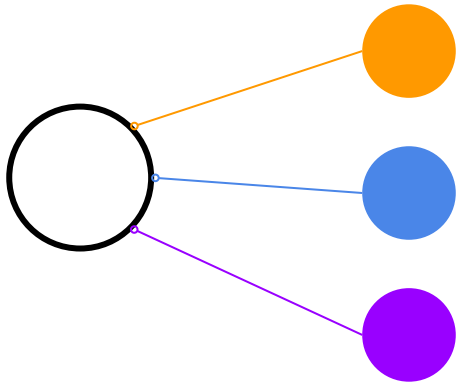
- A *functional program* is like a dictionary
    - Each **function** defines a new concept by expressing a relation between simpler ideas
    - To build a **domain vocabulary** in which the solution of our problem can be **clearly expressed**

redradix

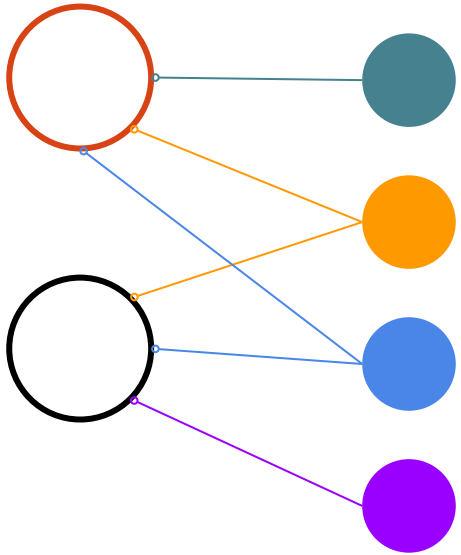


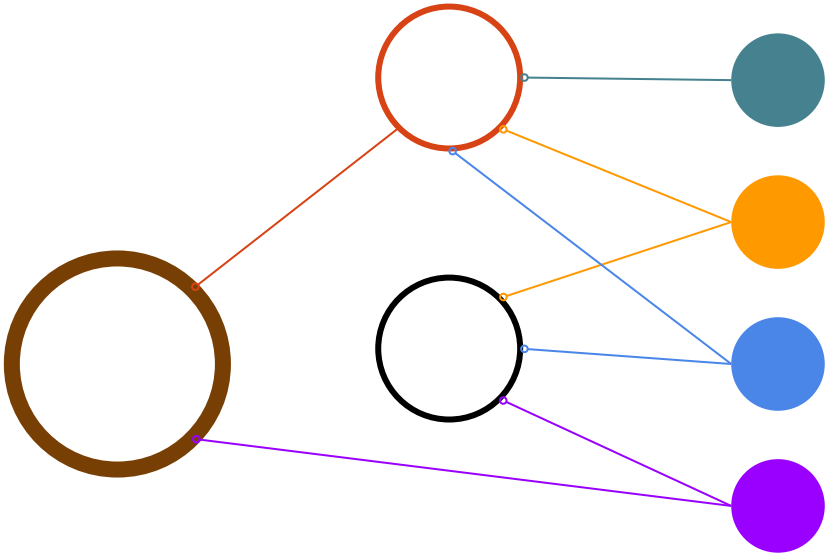
redradix

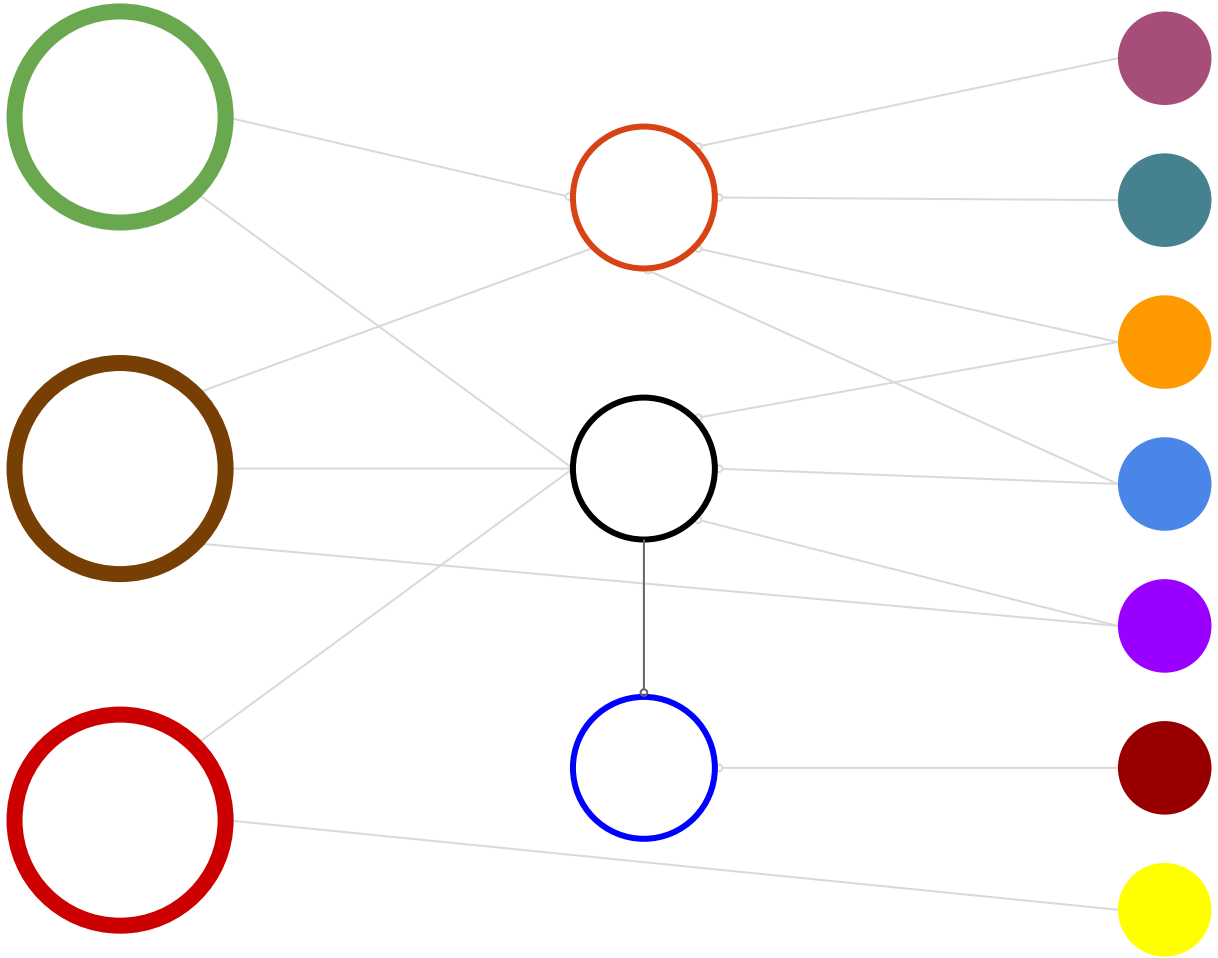
redradix



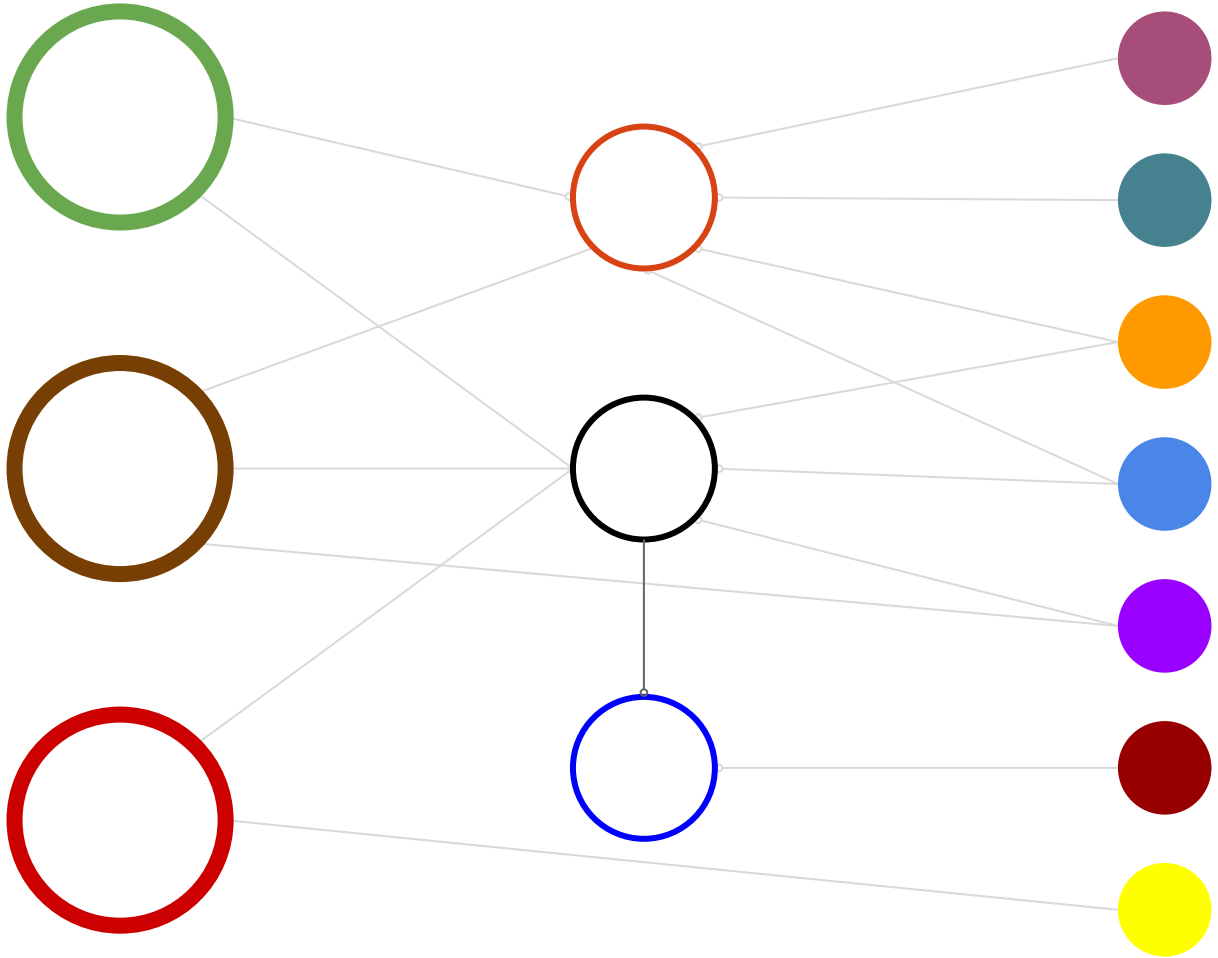
redradix







redradix



redradix

Function Composition

- A functional program is built in a *different way*
  - starting with the tools provided by the language
  - we build many, many small functions
  - that are **combined** to **express** complex ideas of a higher level
  - until we have a **vocabulary rich enough** to **express what we want to achieve**
  - thinking in terms of **results**, not steps

Function Composition

- Very high code reusability
- The program grows in a very organic fashion
    - The bigger it grows, the richer its vocabulary becomes
    - Richer vocabulary makes it easier to keep extending it

Function Composition

- *functional programming*, as a paradigm, exists because **functions can be combined**.

Function Composition

- The functions *want* to be combined!
    - **factorized** in simpler functions
    - become **well defined abstractions** that can be used to express higher level concepts

Function Composition

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

Function Composition

```
const head = list => list[0];
const tail = list => list.slice(1);

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

Function Composition

- Recursion is the simplest form of composition
- Recursion > iteration
    - more expressive
    - **who handles the state?**

Exercise: filterTree

- Implement `filterTree(fn, tree)`
  - **tree**: nested arrays
  - removes the **leaves** for which **fn** returns **false**

Exercise: 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]]]]
```

Exercise: filterTree

- Write a **recursive** implementation
- And then write an **iterative** implementation

Functional vs. Imperative

- A functional program starts *in the algorithm*
    - imperative algorithm ⇨ imperative code
    - functional algorithm ⇨ functional code

Exercise: recursion

- implement **`sumUntil(n)`**
  - Returns the sum of every natural number
    - from 0
    - up to **n**
  - **Don’t** use loops or variables!

Exercise: recursion

- Implement **`strCount(haystack, needle)`**
  - **haystack** and **needle** are strings
  - Counts how many times **needle** appears in **haystack**
  - **Don’t** use loops or variables!

Function Composition

- `compose(fn1, fn2, fn3, …)`
  - Returns a **new function**
  - The classical composition operation
  - Creates new operations using existing functions
  - `compose(a, b)(x) === a(b(x))`

Composición de Funciones

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

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

Exercise: compose

- Implement **`compose(fn1, fn2, ...)`**
  - Every function receives only one parameter
  - Except the **last one**, that can receive any number of parameters

Function Composition

- **compose** is very useful to create specific **configurations** of existing operations
  - used with **map**, **reduce**, **filter**, etc...
- Its usefulness depends on the size of the **library of utilities** we have available to combine

Function Composition

```
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
);
```

Function Composition

- `pipe(fn1, fn2, fn3, …)`
  - Like **compose**, but with the parameters in the inverse order
  - **fn1** runs first and the result goes to **fn2**, etc...
  - `pipe(a, b)(x) === b(a(x))`

Function Composition

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

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

Exercise: pipe

- Implement **`pipe(fn1, fn2, ...)`**
  - As a **composition** of other functions
  - `const pipe = compose(....)`
  - Write any **auxiliary functions** you may need

Function Composition

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

Function Composition

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

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

Exercise: fizzbuzz

- Implement **`fizzbuzz`**
  - As a function **composition**
  - `const fizzbuzz = compose(....)`
  - Write any **auxiliary functions** you may need

Function Composition

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

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

Function Composition

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

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

Function Composition

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

fizzbuzz(100);
```

Function Composition

- `branch(testFn, trueFn, falseFn)`
  - Conditional composition
  - The functional version of the ternary operator

Function Composition

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

const tirada = branch(condition, heads, tails);
```

Exercise: `branch`

- Implement **`branch(test, trueFn, falseFn)`**
  - returns a function
  - when ran, invokes **test**
    - if test returns **true**, runs **trueFn**
    - if test returns **false**, runs **falseFn**

Function Composition

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

Function Composition

- `maybe(fn)`
  - Conditional execution to prevent errors
  - Returns a new function
  - Only runs **fn** if invoked with a *truthy* param or `0`

Function Composition

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

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

Exercise: `maybe`

- Implement **`maybe(fn)`**
  - Implement any utilities you may need

Function Composition

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

Function Composition

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

Function Composition

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

Function Composition

```
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);
```

Function Composition

- **compose**, **branch** and **maybe** are examples of the functional way of thinking:
  - small utilities that express the relation between different elements
  - even if the equivalente imperative code looks similar, using **compose**/**branch**/**maybe** the intent is more clearly expressed

Function Composition

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

. vs .

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

Tacit Programming

Tacit Programming

- Point-free programming
- Define new functions using other functions
- Without making its arguments explicit
- Point-free = without arguments

Tacit Programming

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

redradix

Tacit Programming

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

Tacit Programming

- Max code reusability!
- Encourages **thinking about function composition** (*high level*)...
- ...instead of data manipulation (*low level*)
- Promotes the creation of **rich, orthogonal domain vocabulary**

Tacit Programming

- It’s not a paradigm
- It’s an **ideal style** we should aim to achieve
- it’s the “nirvana” of functional programming
- Sign of a **mature, well designed architecture**
- Found in the higher levels of the program

Exercise: fizzbuzz

- Implement **`fizzbuzz`**
  - using **branch**
  - `const fizzbuzz = branch(....)`
  - Write any auxiliary functions you may need
    - try to create easy to compose utilities
    - to achieve compact, clear code

Tacit Programming

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

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

Tacit Programming

```
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);
```

Tacit Programming

- When writing functional code...
    - It's important to **be alert**
    - to detect **when we are missing a new utility**
    - to **express our algorithm better**

Exercise: fizzbuzz

- Implement **`fizzbuzz`** again
  - creating **a new utility**
  - to express the logic better than **branch**
  - think this: “*What would be the best way to express this algorithm?*”

Tacit Programming

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

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

Tacit Programming

```
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));
```

Tacit Programming

```
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);
}
```