FUNCTIONAL MAGIC

Neem Serra
@TeamNeem

Functional Programming

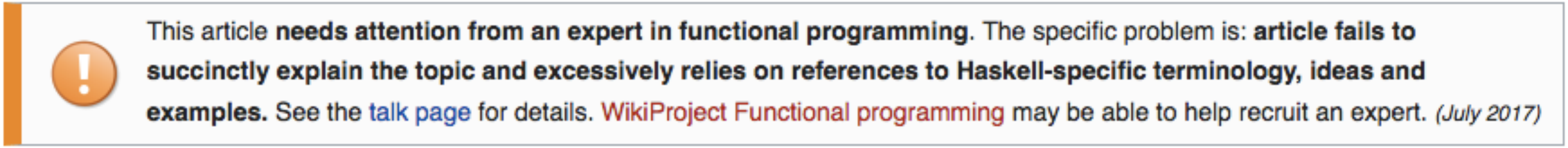
Functional Programming

$\downarrow$

Map, FlatMap, CompactMap, Filter, Reduce

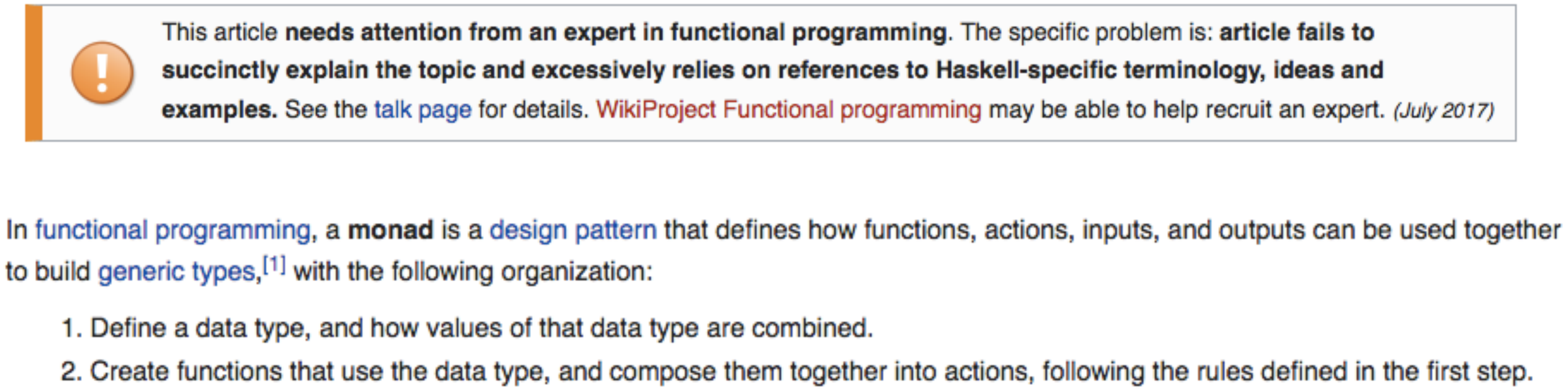
$$\begin{tikzcd}
\text{Functional Programming} \arrow[d] \\
\text{Map, FlatMap, CompactMap, Filter, Reduce} \arrow[d] \\
\text{Monads}
\end{tikzcd}$$

Monads



This article **needs attention from an expert in functional programming**. The specific problem is: **article fails to succinctly explain the topic and excessively relies on references to Haskell-specific terminology, ideas and examples.** See the talk page for details. WikiProject Functional programming may be able to help recruit an expert. *(July 2017)*

Monads

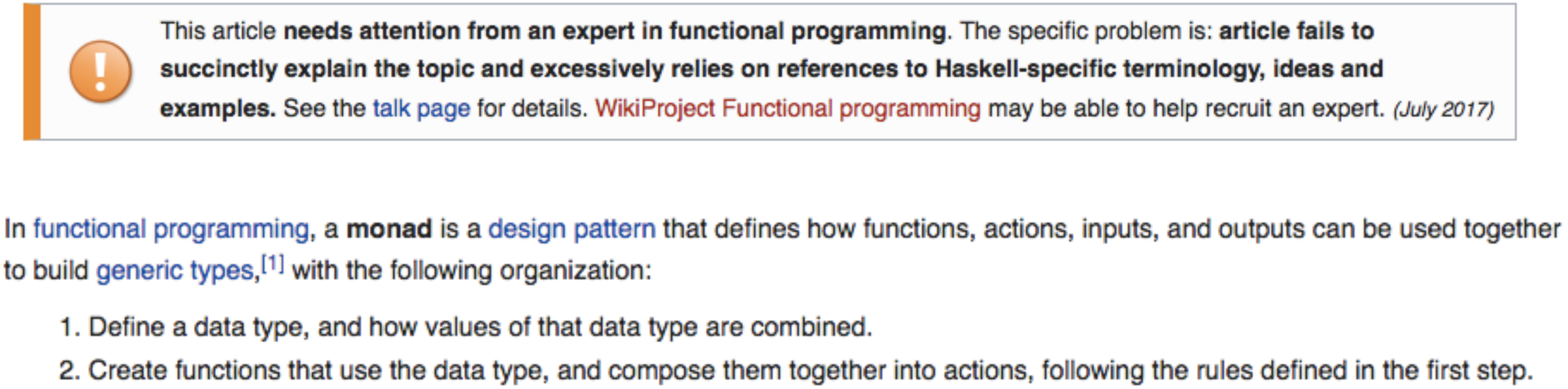


This article **needs attention from an expert in functional programming**. The specific problem is: **article fails to succinctly explain the topic and excessively relies on references to Haskell-specific terminology, ideas and examples.** See the talk page for details. WikiProject Functional programming may be able to help recruit an expert. *(July 2017)*

In functional programming, a **monad** is a design pattern that defines how functions, actions, inputs, and outputs can be used together to build generic types,[1] with the following organization:

1. Define a data type, and how values of that data type are combined.
2. Create functions that use the data type, and compose them together into actions, following the rules defined in the first step.

Monads



This article **needs attention from an expert in functional programming**. The specific problem is: **article fails to succinctly explain the topic and excessively relies on references to Haskell-specific terminology, ideas and examples.** See the talk page for details. WikiProject Functional programming may be able to help recruit an expert. *(July 2017)*

In functional programming, a **monad** is a design pattern that defines how functions, actions, inputs, and outputs can be used together to build generic types,[1] with the following organization:

1. Define a data type, and how values of that data type are combined.
2. Create functions that use the data type, and compose them together into actions, following the rules defined in the first step.

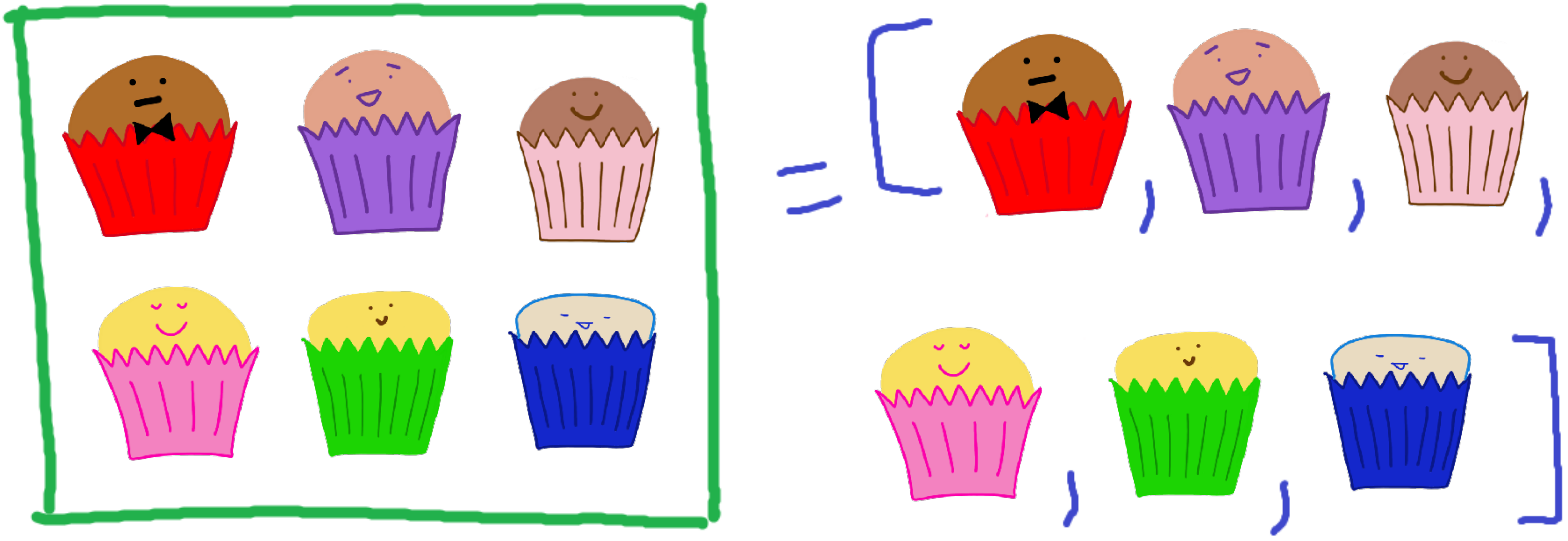
$$\begin{tikzcd}
\text{Functional Programming} \arrow[d] \\
\text{Map, FlatMap, CompactMap, Filter, Reduce} \arrow[d] \\
\text{Monads}
\end{tikzcd}$$

Overwhelmed

But it is okay!

Functional Magic via

Cupcakes!



Cupcake

Attributes:

```
    var frosting: Frosting?
    var cake: Cake
```

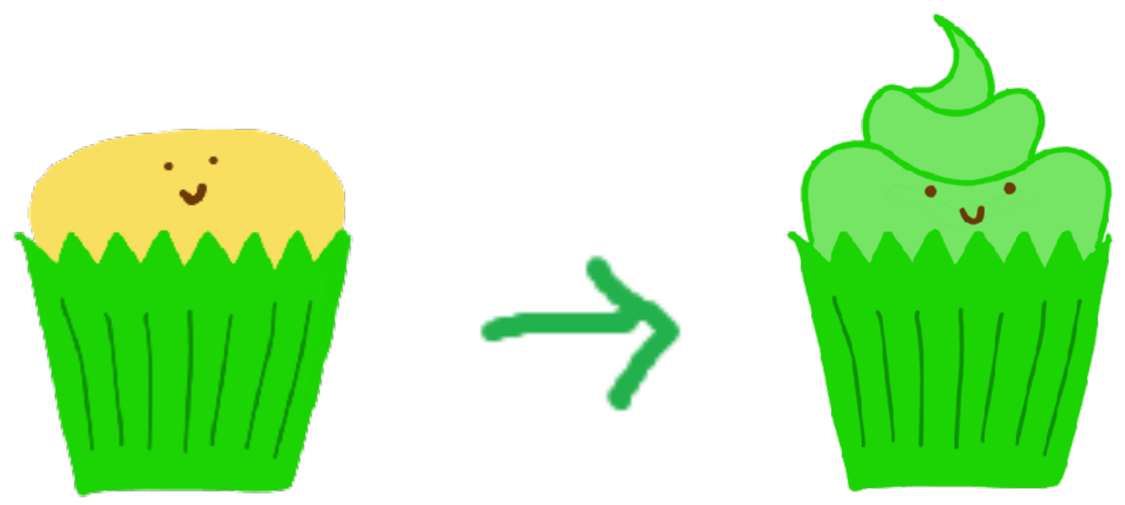
Cupcake

Attributes:

```
var frosting: Frosting?
var cake: Cake
```

Functions:

```
func frosted() -> Cupcake
```



```
func getFlavor() -> String
```

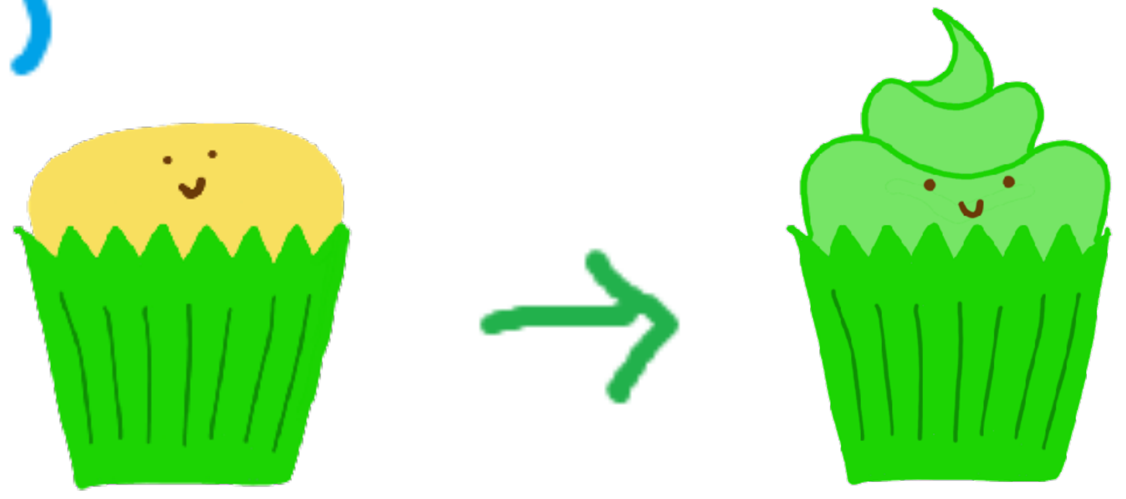
Map

Loop over a collection and apply the same operation to each element

let cupcakes = [ ]

Old Way:

```
for cupcake in cupcakes {
    let frostedCupcake = cupcake.frosted()
}
```

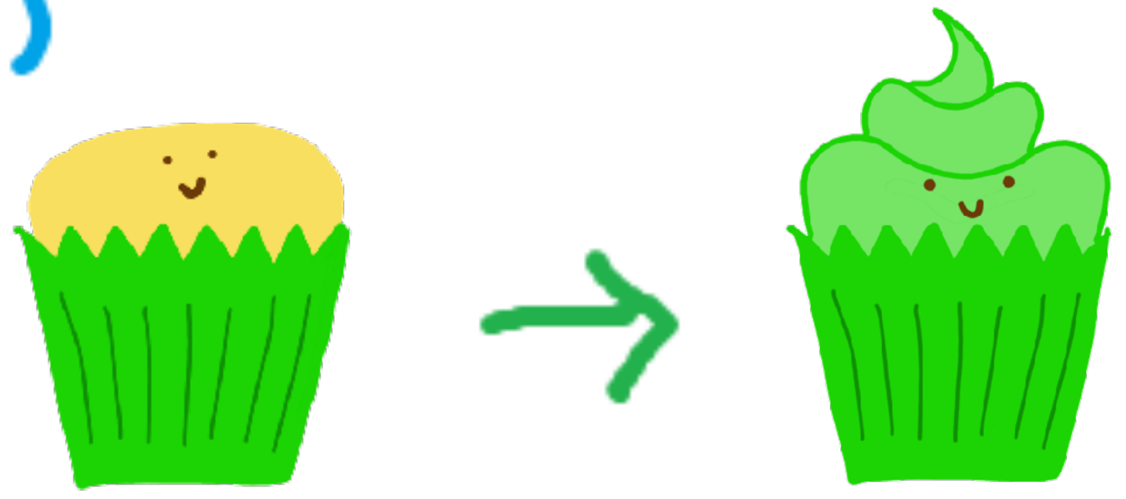


```
let cupcakes = [
```

```
]
```

Old Way:

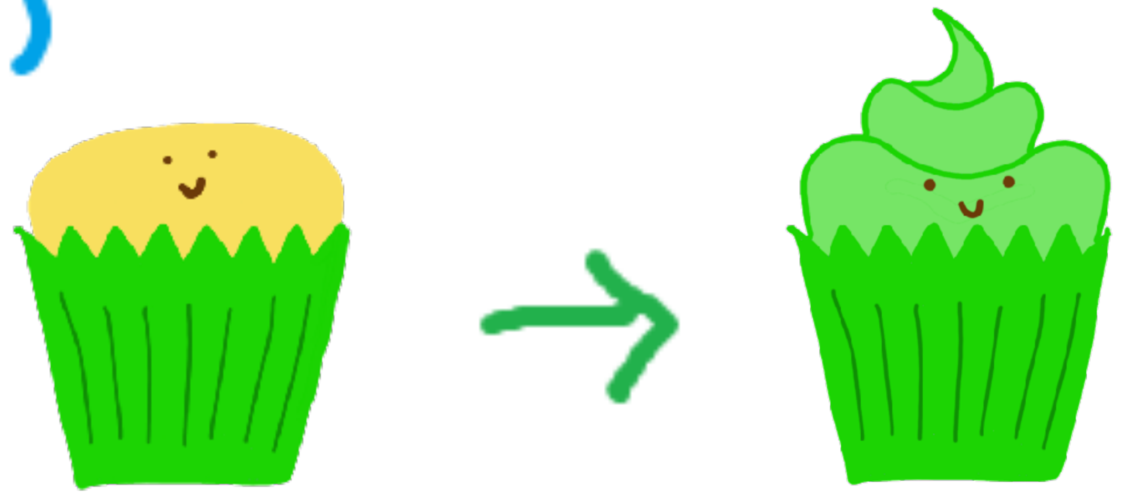
```
for cupcake in cupcakes {
    let frostedCupcake = cupcake.frosted()
}
```



let cupcakes = [ ]

Old Way:

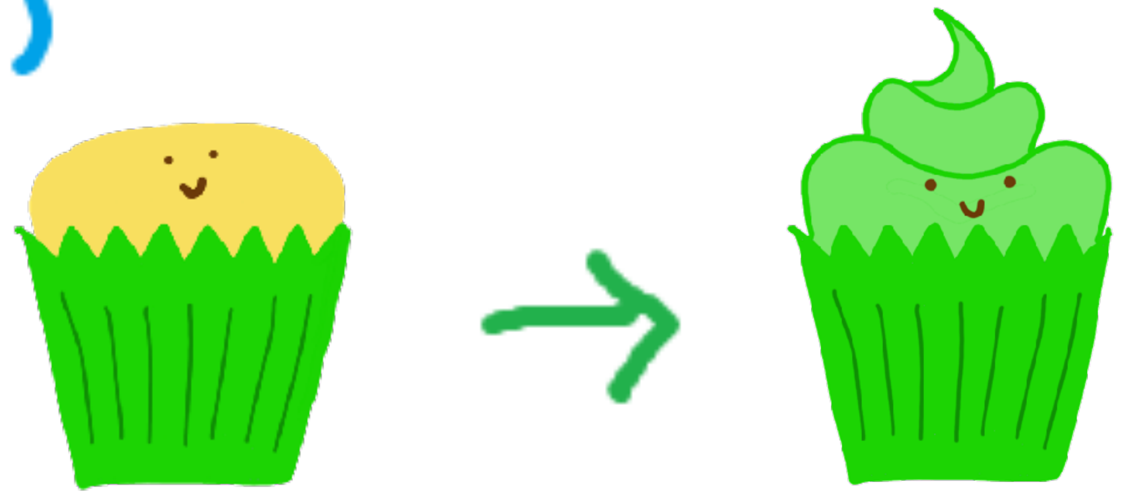
```
for cupcake in cupcakes {
    let frostedCupcake = cupcake.frosted()
}
```



let cupcakes = [ ]

Old Way:

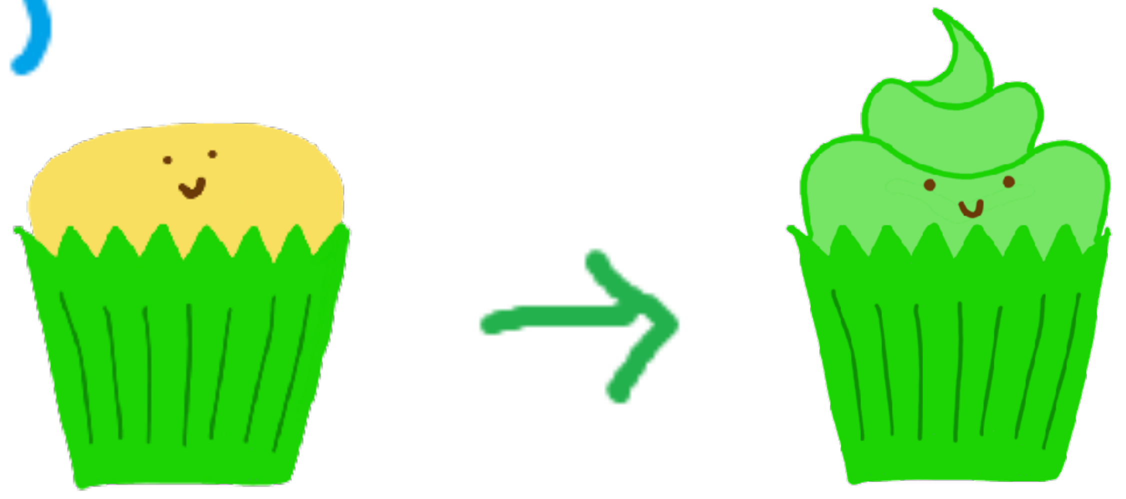
```
for cupcake in cupcakes {
    let frostedCupcake = cupcake.frosted()
}
```



let cupcakes = [ ]

Old Way:

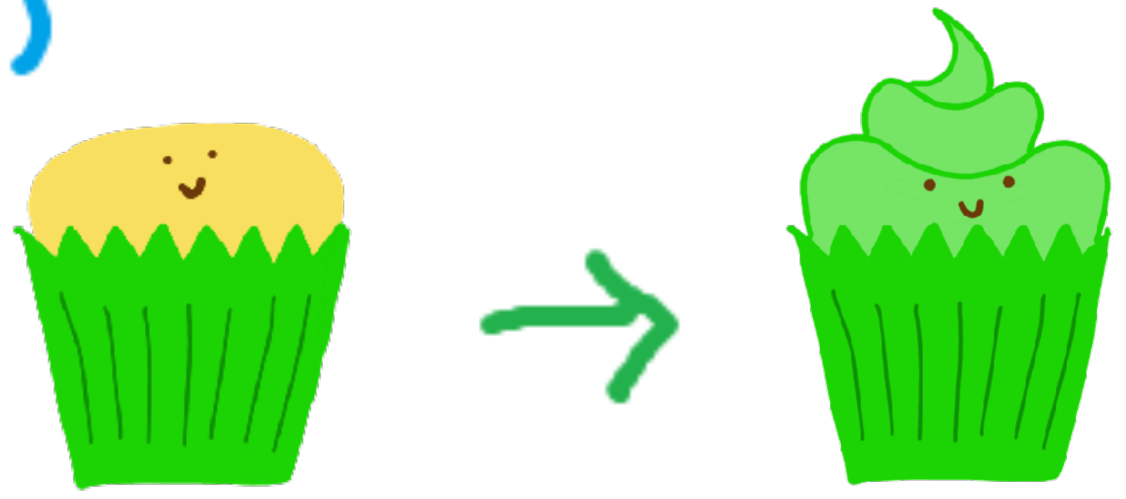
```
for cupcake in cupcakes {
    let frostedCupcake = cupcake.frosted()
}
```



let cupcakes = [ ]

Old Way:

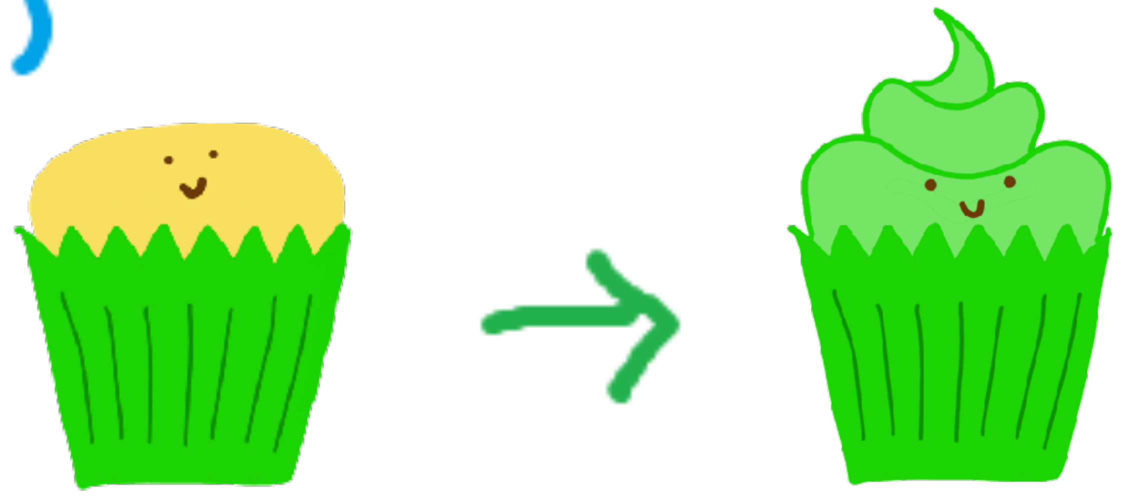
```
for cupcake in cupcakes {
    let frostedCupcake = cupcake.frosted()
}
```

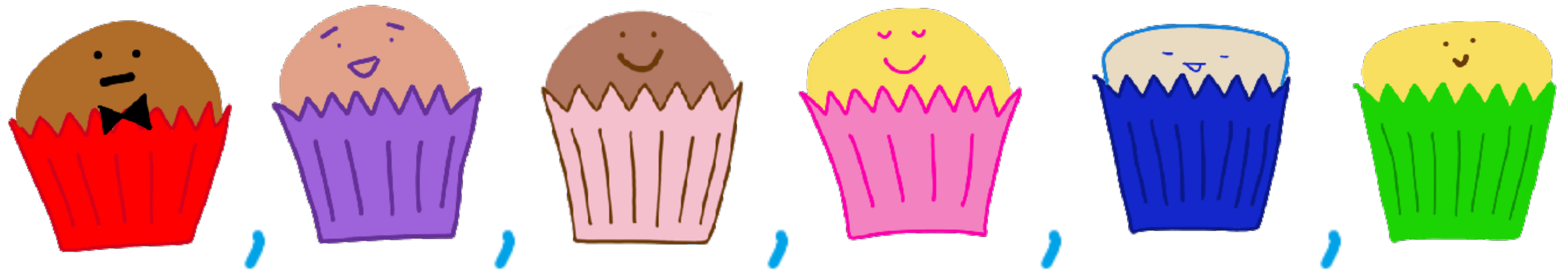


let cupcakes = [ ]

Old Way:

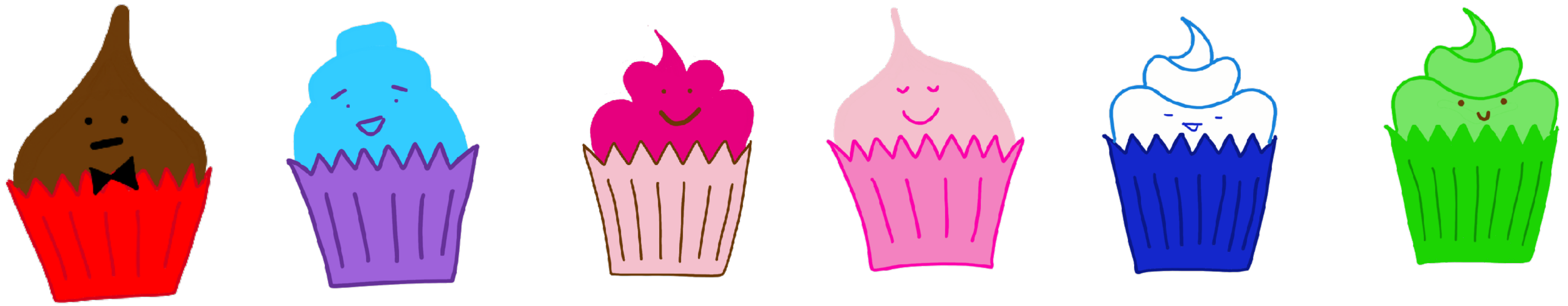
```
for cupcake in cupcakes {
    let frostedCupcake = cupcake.frosted()
}
```



let cupcakes = []

New way:

```
let frostedCupcakes = cupcakes.map{$0.frosted()}
```



let cupcakes = [ ]

Old Way:

```
var cupcakeFlavors :[String] = []
for cupcake in cupcakes {
      let cupcakeFlavor = cupcake.getFlavor()
      cupcakeFlavors.append(cupcakeFlavor)
}
```

let cupcakes = [

, , , , ,]

New Way:

```
let cupcakeFlavors = cupcakes.map{$0.getFlavor()}
```

```
["Red", "22 Baby Blue", "WelcomeToNewYorkMagenta",
"LoveStoryPink", "Style", "ShakeItOff Green"]
```

Old Way:

```
var cupcakeFlavors :[String] = []
for cupcake in cupcakes {
     let cupcakeFlavor = cupcake.getFlavor()
     cupcakeFlavors.append(cupcakeFlavor)
 }
```

New Way:

```
let cupcakeFlavors = cupcakes.map{$0.getFlavor()}
```

Syntax for days ...

```
cupcakes.map({(cupcake: Cupcake) -> Cupcake in
        cupcake.frosted()})
cupcakes.map{(cupcake: Cupcake) in
        cupcake.frosted()}
cupcakes.map { cupcake in
        cupcake.frosted()}
Cupcakes.map { $0 .frosted() }
```

All the same

Map

* Takes a single argument of a closure
* The closure takes the element from the collection and returns the result
* The results are then returned as an array of the closure return type

What about dictionaries?

* Each element is a tuple of key, value
* There is no sense of ordering

Map with Dictionaries

let cupcakeDictionary = [: "Red",
: "22BabyBlue", : "Style"]

let flavors = cupcakeDictionary.map{$1}

Map with Dictionaries

let cupcakeDictionary = [: "Red",
: "22BabyBlue", : "Style"]

let flavors = cupcakeDictionary.map{$1}

["Red", "Style", "22BabyBlue"]

Not in the same order

Mapping with Optionals

```
let numberOfCupcakes: Int? = 0
let newCount = numberOfCupcakes + 2
```

Mapping with Optionals

```
let numberOfCupcakes: Int? = 0
let newCount = numberOfCupcakes + 2
```

Error! Value of type optional not unwrapped

Mapping with Optionals

```
let numberOfCupcakes: Int? = 0
let newCount = numberOfCupcakes + 2
```

Error! Value of type optional not unwrapped

```
let newCount = numberOfCupcakes.map {$0+2}
```

$\Rightarrow$ Optional(2)

FlatMap

* "Flattens" the container then maps over the array
* Sees things in one less dimension

FlatMap

Flattens a collection of collections

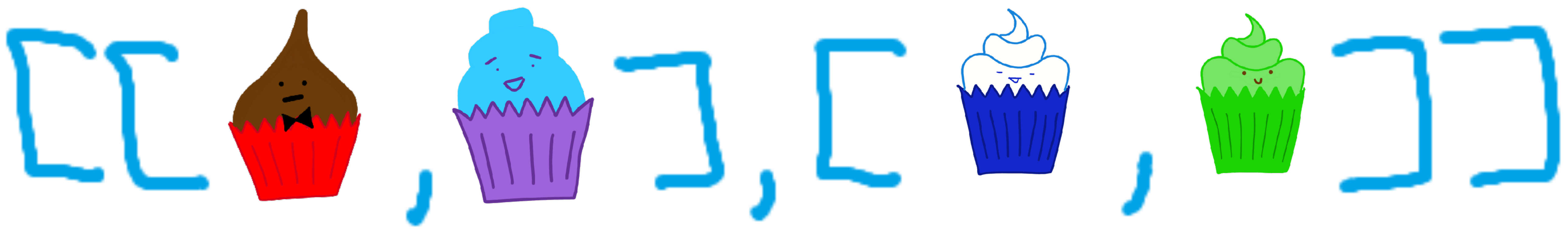
let cupcakeBoxes = [[,], [,]]

```
let newCupcakeBox = cupcakeBoxes.flatMap { $0 }
```

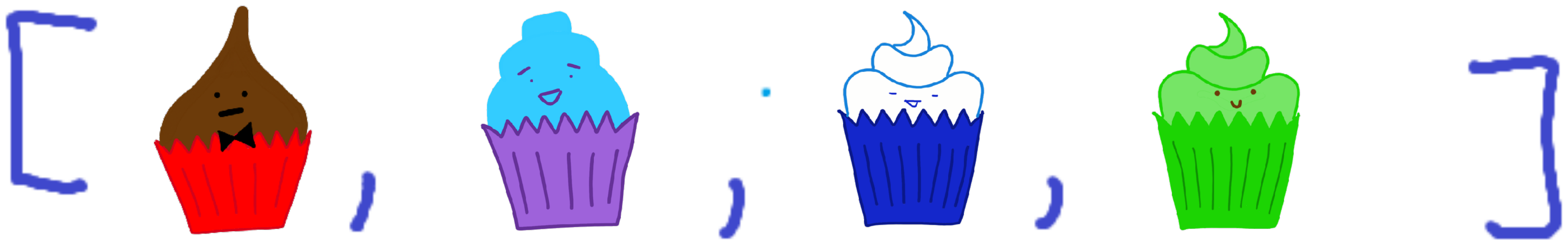
FlatMap

Flattens a collection of collections

let cupcakeBoxes =



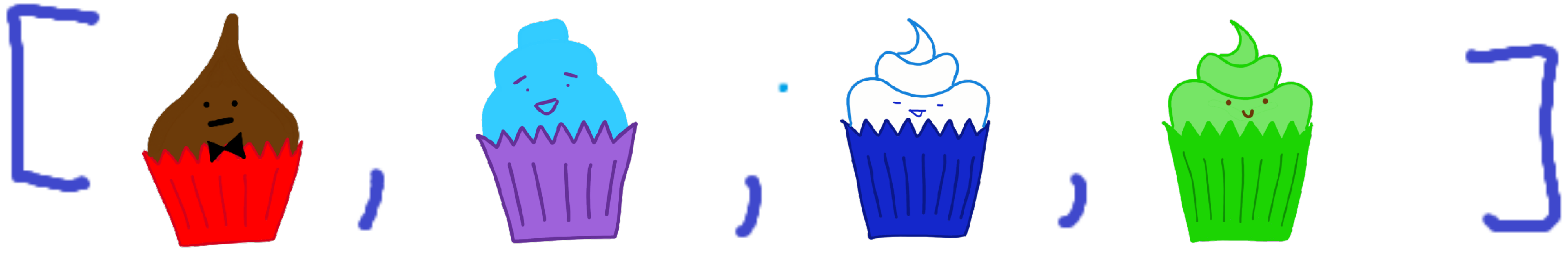
```
let newCupcakeBox = cupcakeBoxes.flatMap{$0}
```



Old Way:

let cupcakeBoxes = [[,], [,]]

```
var newCupcakeBox : [Cupcake] = []
for box in cupcakeBoxes {
    newCupcakeBox += box
}
```



FlatMap

```
let cupcakeNumbers=[[3,4],[1,3]]
```

What is $0?

FlatMap

let cupcakeNumbers = [[3 , 4], [1 , 3]]

What is $0? Each array

FlatMap

```
let cupcakeNumbers=[[3,4],[1,3]]
```

What is $0? Each array

```
cupcakeNumbers.flatMap{$0*2}
```

FlatMap

```
let cupcakeNumbers = [[3, 4], [1, 3]]
```

What is $0? Each array

```
cupcakeNumbers.flatMap {$0 * 2}
```

Error!

FlatMap

```
let cupcakeNumbers = [[3, 4], [1, 3]]
```

What is $0? Each array

```
cupcakeNumbers.flatMap {$0 * 2}
```

Error!

```
let double = cupcakeNumbers.flatMap {array in
    array.map {number in
        number * 2
    }
}
```

FlatMap

let cupcakeNumbers=[[3 , 4],[1 , 3]]

What is $0? Each array

cupcakeNumbers.flatMap {$0 *2} Error!

```
let double = cupcakeNumbers.flatMap {array in
      array.map { number in
           number * 2
      }
}
```

Map
[6, 8], [2, 6]

FlatMap
[6, 8, 2, 6]

FlatMap

* Takes a single argument of a closure
* The closure takes the element from the collection and flattens the collection
* The results are then returned as an array of the closure return type

CompactMap

Convenient way to strip arrays of nil values and unwrap optionals

CompactMap

Deals with optionals. Deprecated flatMap

CompactMap

Deals with optionals. Deprecated flatMap

```
let cupcakes:[Cupcake?] = [
```

```
, nil,
```

```
, nil]
print (cupcakes.map {$0})
[Optional(
```

```
), nil, Optional(
```

```
), nil]
```

Compact Map

Deals with optionals. Deprecated flatMap

```
let cupcakes: [Cupcake?] = [
```

```
, nil,
```

```
, nil]
print (cupcakes.map {$0})
[Optional(
```

```
), nil, Optional(
```

```
), nil]
print (cupcakes.CompactMap {$0})
[
```

```
,
```

```
]
```

Old Way:

```
let realCupcakes: [Cupcake] = []
for cupcake in cupcakes {
    if let cupcake = cupcake {
        realCupcakes.append(cupcake)
    }
}
```

[,]

CompactMap

* Takes a single argument of a closure
* The closure takes the element from the collection and removes nils / unwraps optionals
* The results are then returned as an array of the closure return type

Filter

Loop over a collection and return an array that has elements that match an include condition

Filter

Only return cupcakes that are not red

old way:

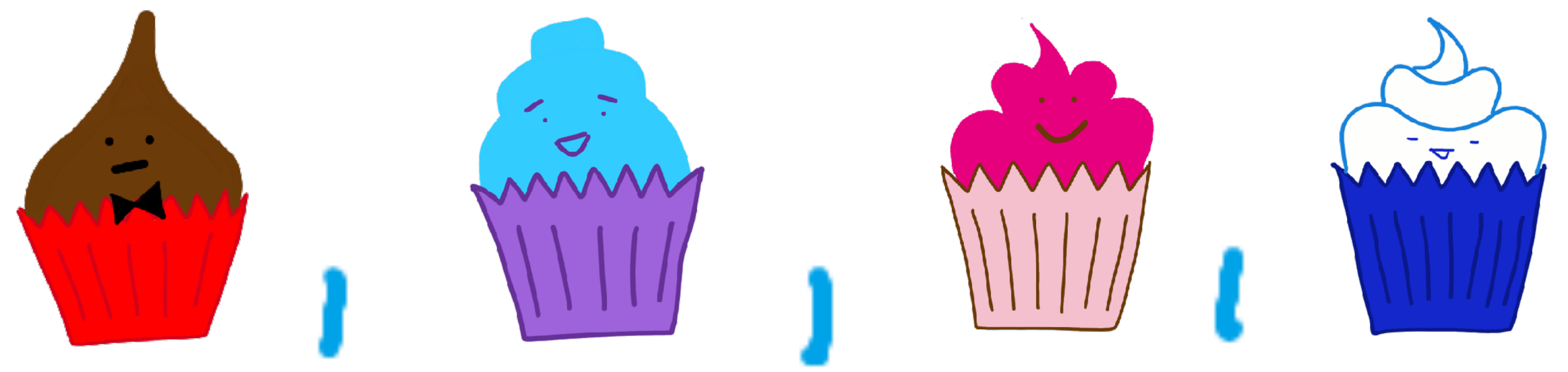
let cupcakes = [, , ,]

```
let newBox : [Cupcake] = []
for cupcake in cupcakes {
        if cupcake.getFlavor != "Red" {
            newBox.append(cupcake)
        }
}
```

Filter

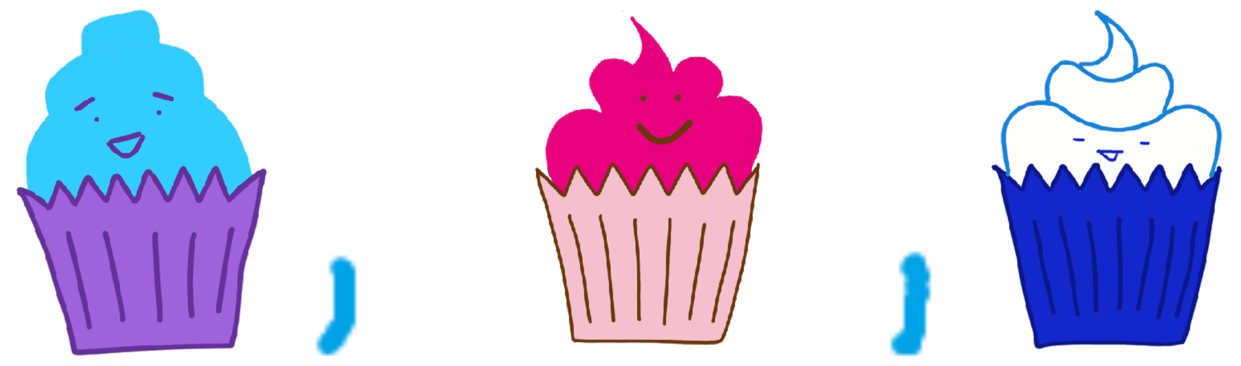
New way:

```
let cupcakes = [ , , , ]
```



```
let newBox = cupcakes.filter {
        $0.getFlavor != "Red"
}
```

```
[ , , ]
```



Filter

* Takes a single argument of a closure
* The closure takes the element from the collection and determines if included
* The results are then returned as an array of the closure return type

Reduce

Combine all items in a collection into a single value

Reduce

```
reduce(initialValue) {result, nextItem in
    return result + nextItem }
reduce(initialValue) { $0 + $1 }
reduce(initialValue) { $0, + }
```

Reduce

Get final price of cupcakes

Old way:

let cupcakes = [, , ,]

```
var finalPrice = 3
for cupcake in cupcakes {
    finalPrice += 2
}
```

Reduce

New way:

let cupcakes = [, , ,]

let finalPrice = cupcakes.reduce(3) {$0+2}

11

Reduce

New way:

let cupcakes = [, , ,]

```
let finalPrice = cupcakes.reduce(3) {
    $1.frosting ? $0 + 4 : $0 + 2
}
```

15

Reduce

* Takes 2 arguments: initial value and a closure with two arguments – the initial value / previous result, Collection item
* The closure takes the element from the collection combines it with the result
* A single return value is produced

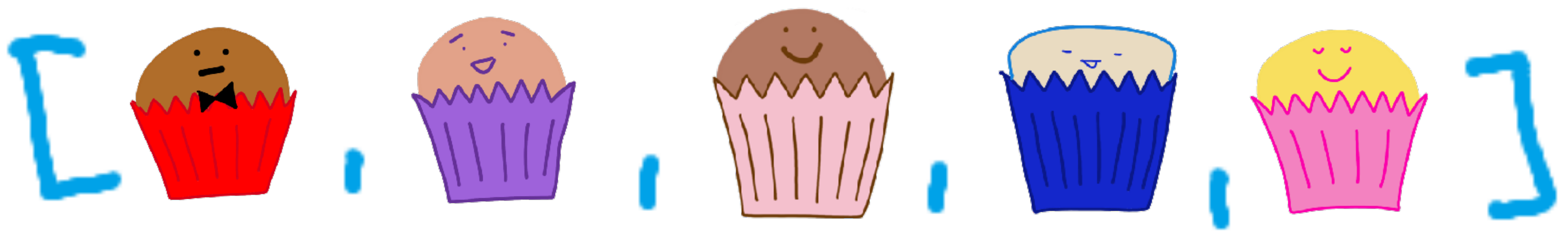
Why do we like these functions?

Visually Explicit Chaining

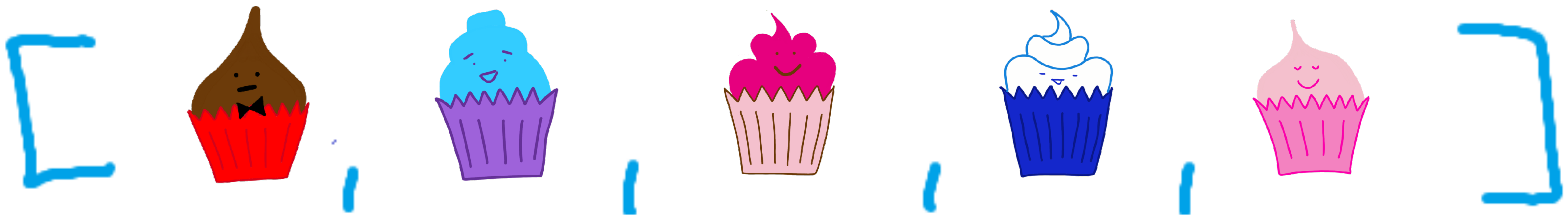
Let's See It All Together

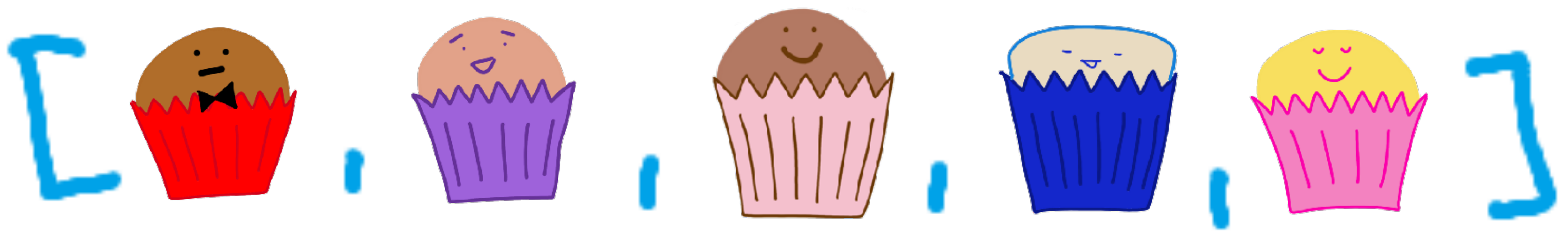
1. Frost one box of cupcakes
2. Remove nils/optionals from second box
3. Combine boxes
4. Filter out cupcakes that are Red flavored
5. Find out final price of unfrosted cupcakes with more expensive frosted cupcakes

`let box1 : [Cupcake] =`

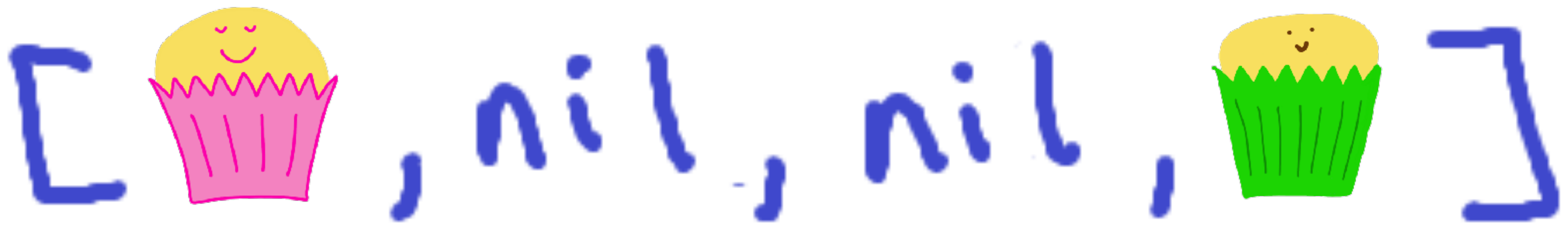


```
let frosted = box1.map { $0.frosted() }
```

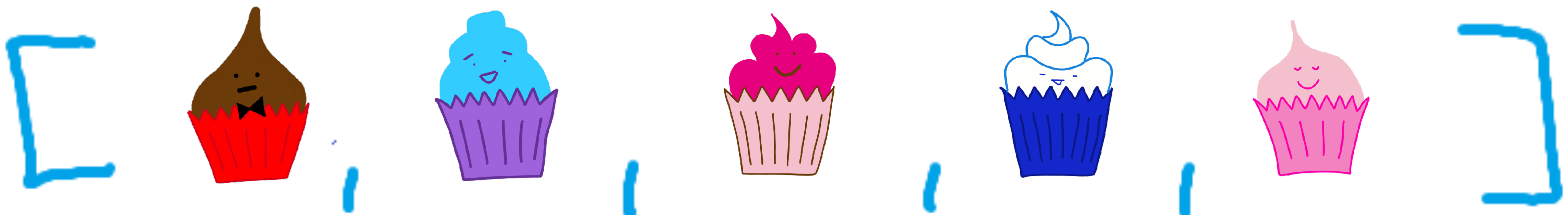


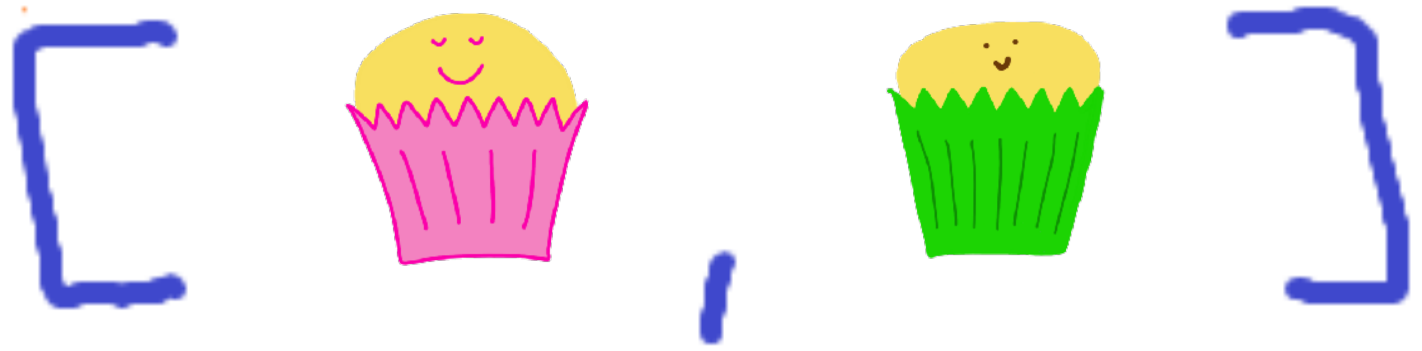
let box1 : [Cupcake] = 

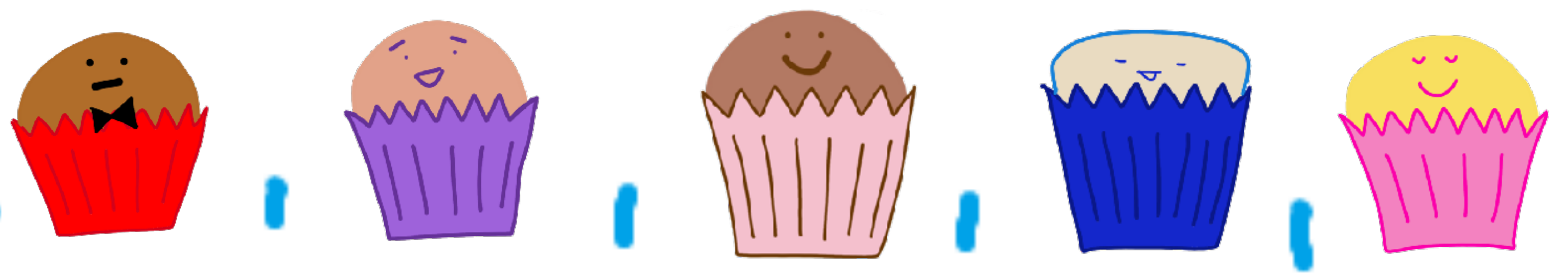
let frosted = box1.map { $0.frosted() }

let box2 : [Cupcake?] = 

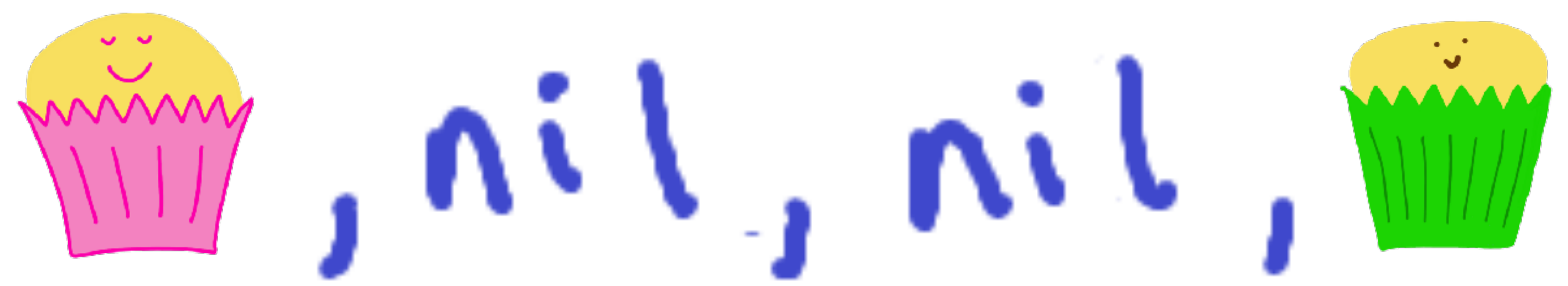
let realCupcakes = box2.compactMap { $0 }



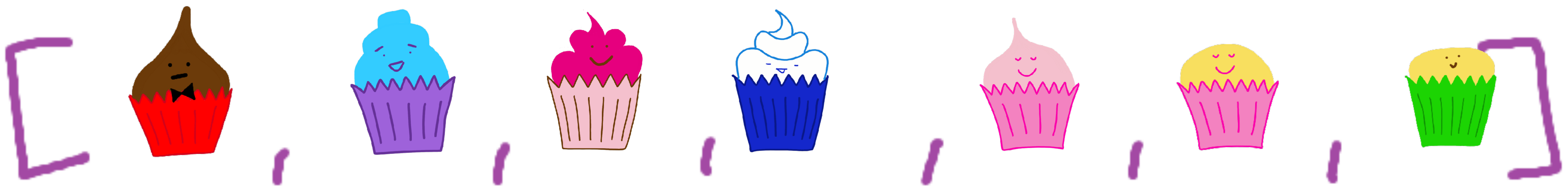


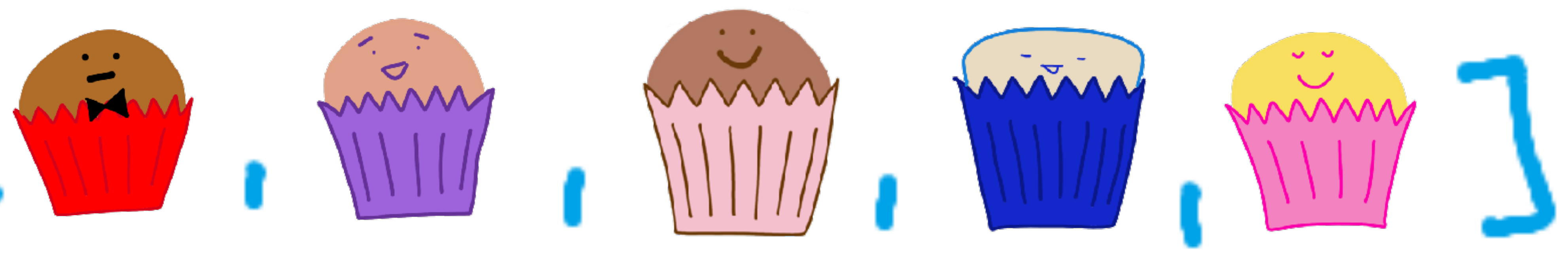
`let box1 : [Cupcake] = [`  `]`

```
let frosted = box1.map { $0.frosted() }
```

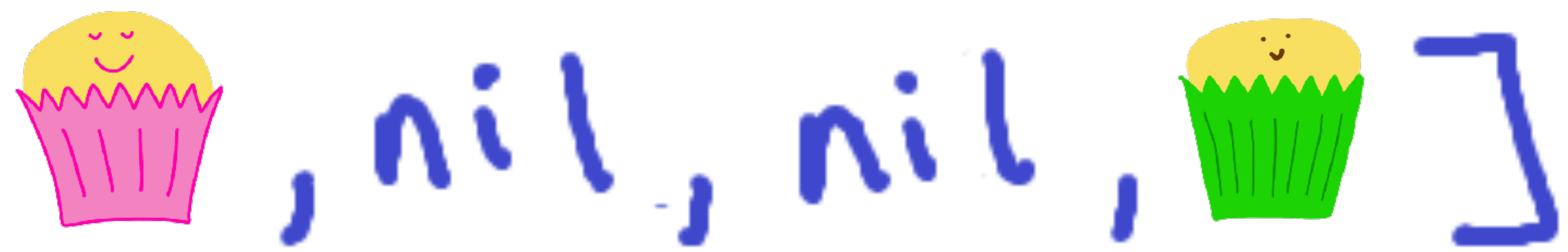
`let box2 : [Cupcake?] = [`  `]`

```
let realCupcakes = box2.compactMap { $0 }
let combined = [frosted, realCupcakes].flatMap { $0 }
```



let box1 : [Cupcake] = [

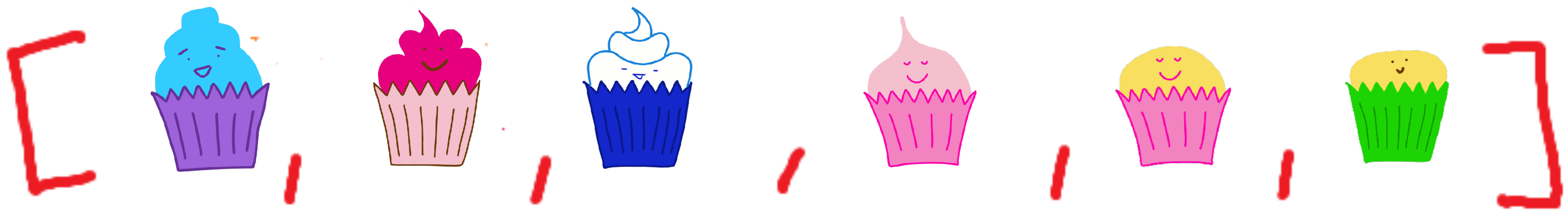
let frosted = box1.map { $0.frosted() }

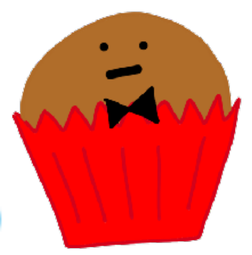
let box2 : [Cupcake?] = [

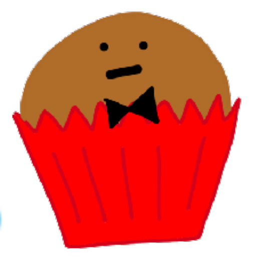
let realCupcakes = box2.compactMap { $0 }

let combined = [frosted, realCupcakes].flatMap {$0}

let noReds = combined.filter { $0.getFlavor() != "Red"}



```
let box1 : [Cupcake] = [, , , , ]
let frosted = box1.map { $0.frosted() }
let box2 : [Cupcake?] = [ , nil, nil, ]
let realCupcakes = box2.compactMap { $0 }
let combined = [frosted, realCupcakes].flatMap {$0}
let noReds = combined.filter { $0.getFlavor() != "Red"}
let finalPrice = noReds.reduce(0) {
        $1.frosting ? $0+4 : $0+2
}
```

```
let box1 : [Cupcake] = [, , , , ]
let frosted = box1.map { $0.frosted() }
let box2 : [Cupcake?] = [ , nil, nil, ]
let realCupcakes = box2.compactMap { $0 }
let combined = [frosted, realCupcakes].flatMap{$0}
let noReds = combined.filter { $0.getFlavor() != "Red" }
let finalPrice = noReds.reduce (0) {
            $1.frosting ? $0+4 : $0 +2
}
```

20

Map: Apply the same function to each item in a collection

FlatMap: Same as map but flatten collection

CompactMap: Same as map but also remove nils and unwrap optionals

Filter: Only return items from a collection that match an include statement

Reduce: Combine items in a collection into a single value

Thank You!!!

Questions?

@TeamNeem