Functional Programming

Better than Dysfunctional Programming


FUNCTIONAL PROGRAMMING
SO HOT RIGHT NOW
imgflip.com

paradigm

noun | par·a·digm | \ˈper-ə-ˌdīm , ˈpa-rə- *also* -ˌdim\

"…3. a philosophical and theoretical framework of a scientific school or discipline…"

MERRIAM WEBSTER

- **Broad categories of programming languages**
- **Traditionally viewed as competing styles** 🤔
- **May have very different syntax, capabilities, goals, and concepts**

(Some) Oft-Cited Examples

Paradigm	Languages
Procedural	FORTRAN, ALGOL, COBOL, C, BASIC
Object Oriented	Simula, Smalltalk, Self, C++, Java, Ruby, Python
Functional	Lisp, Scheme, OCaml, Haskell, F#, Elm, ReasonML
Declarative	SQL, HTML, RegEx, Wolfram

But Wait, There's More!™

Lazily Evaluated
Purely Functional
Structured
Concatenative
Procedural
Stateful
Logic
Concurrent
Markup
Toy
Eagerly Evaluated
Esoteric
Imperative
Statically / Dynamically Typed
Reactive
Strongly / Weakly Typed
Impure Functional
Symbol
Proof Systems

functional programming

functional programming

FP in a 🌰 Nutshell

- 🎶 Functions everywhere (naturally)
- 🎵 Composition of functions (small pieces → larger constructs)
- 💘 Pure functions only (input → function → output, no effects)
- 💞 Equational reasoning / referential transparency (easier to use)
- 💟 First-class / higher-order functions (code uses / produces code)
- 💗 Currying and partial application (general-purpose → specific)
- 💎 Immutable data (foolproof, supports equational reasoning)
- λ Mathematical foundations (lambda calculus, category theory)

FP 💡 Motivations

Feature(s)	Benefit(s)
Many functions, composition, higher-order, currying	Seamlessly derive new code from old, maximum interop btw. program pieces
Pure functions, immutability, no side effects, no state mutation	Equational reasoning, reduce mental scope, make bugs impossible, enable optimizations
Mathematical underpinnings (lambda calculus, category theory)	Universal concepts, provable approaches, static analysis tools, clever tech

James Iry
@jamesiry

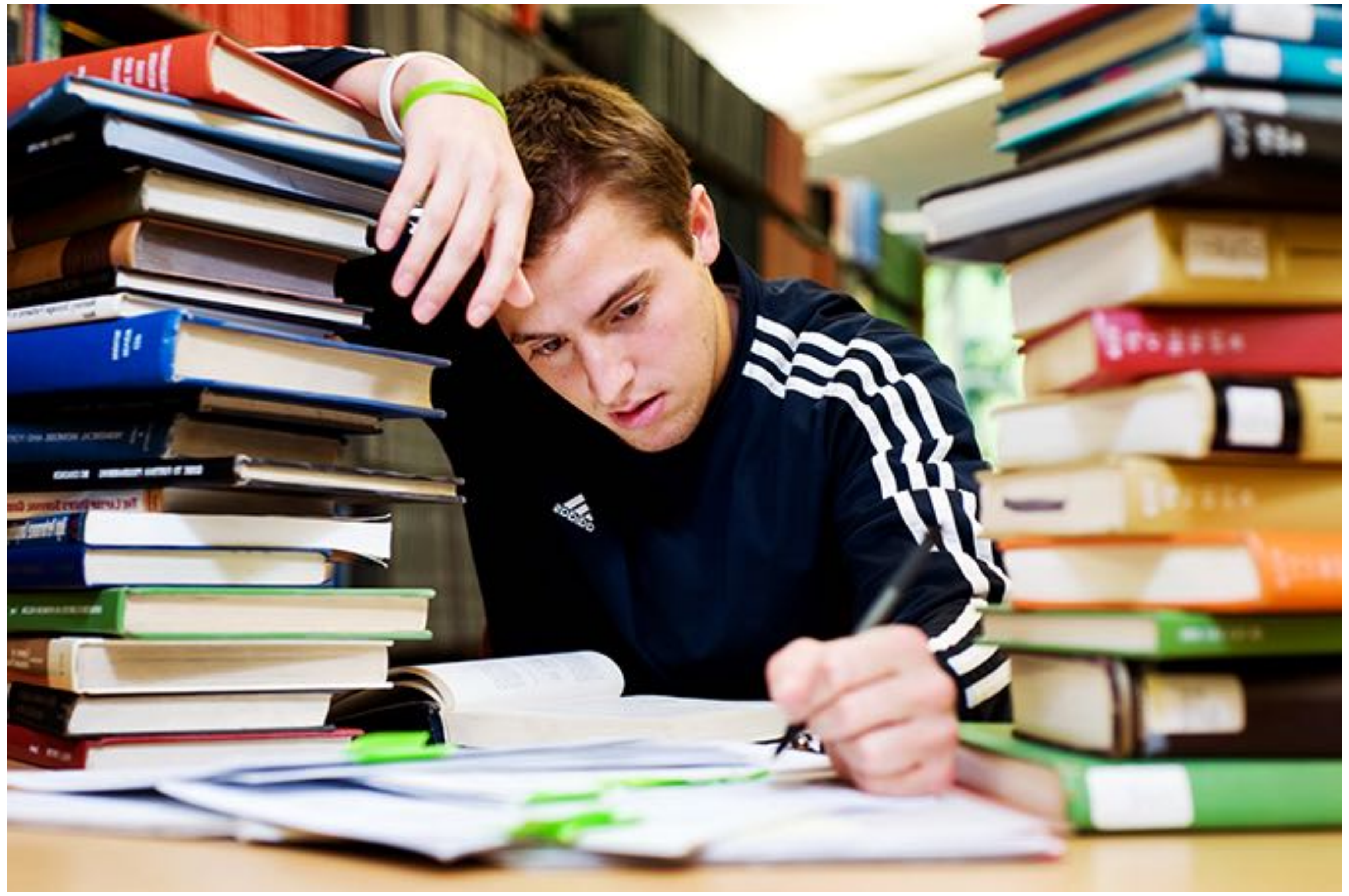
Follow

Functional programmer: (noun) One who names variables "x", names functions "f", and names code patterns "zygohistomorphic prepromorphism"

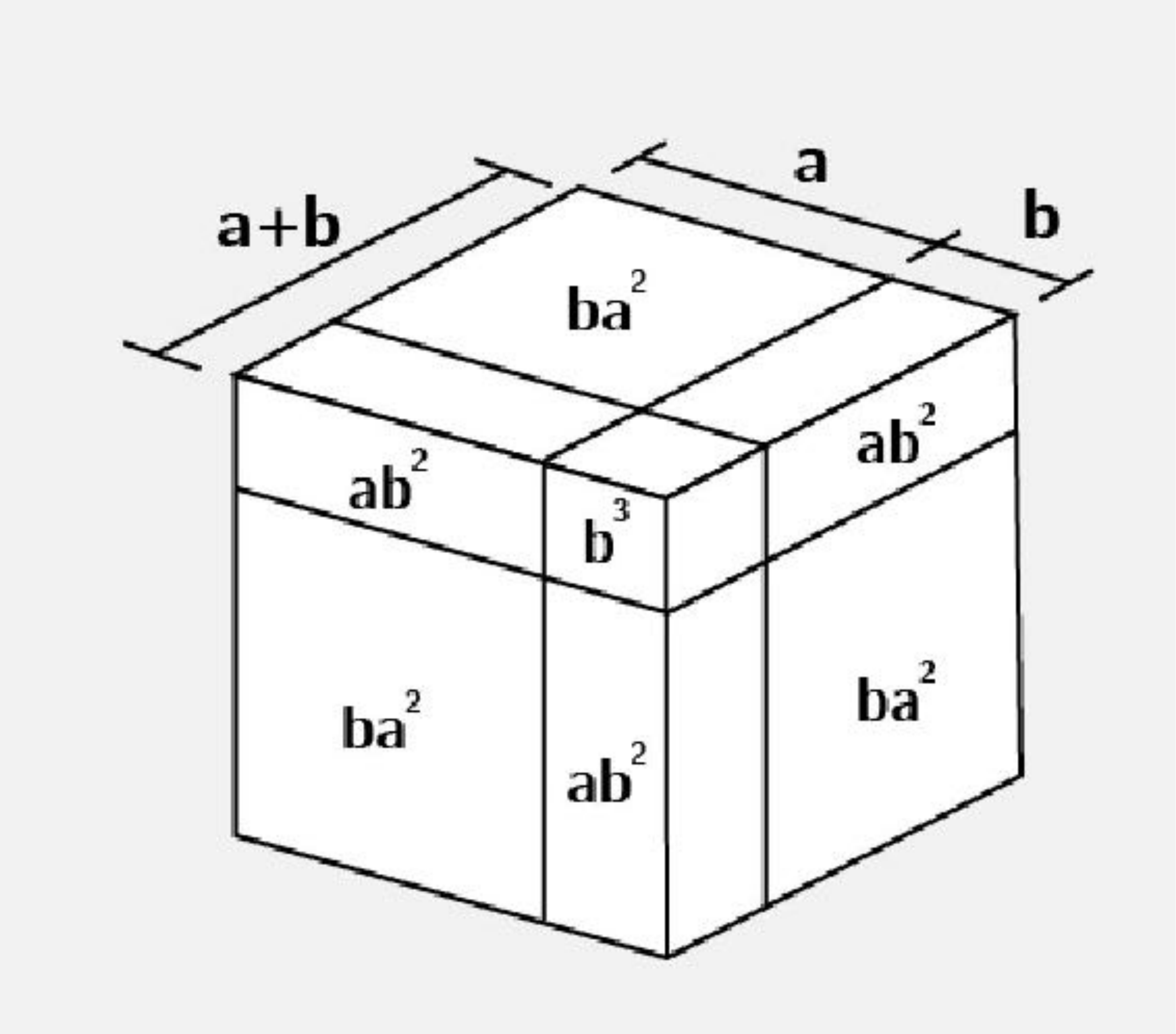
10:58 AM - 13 May 2015

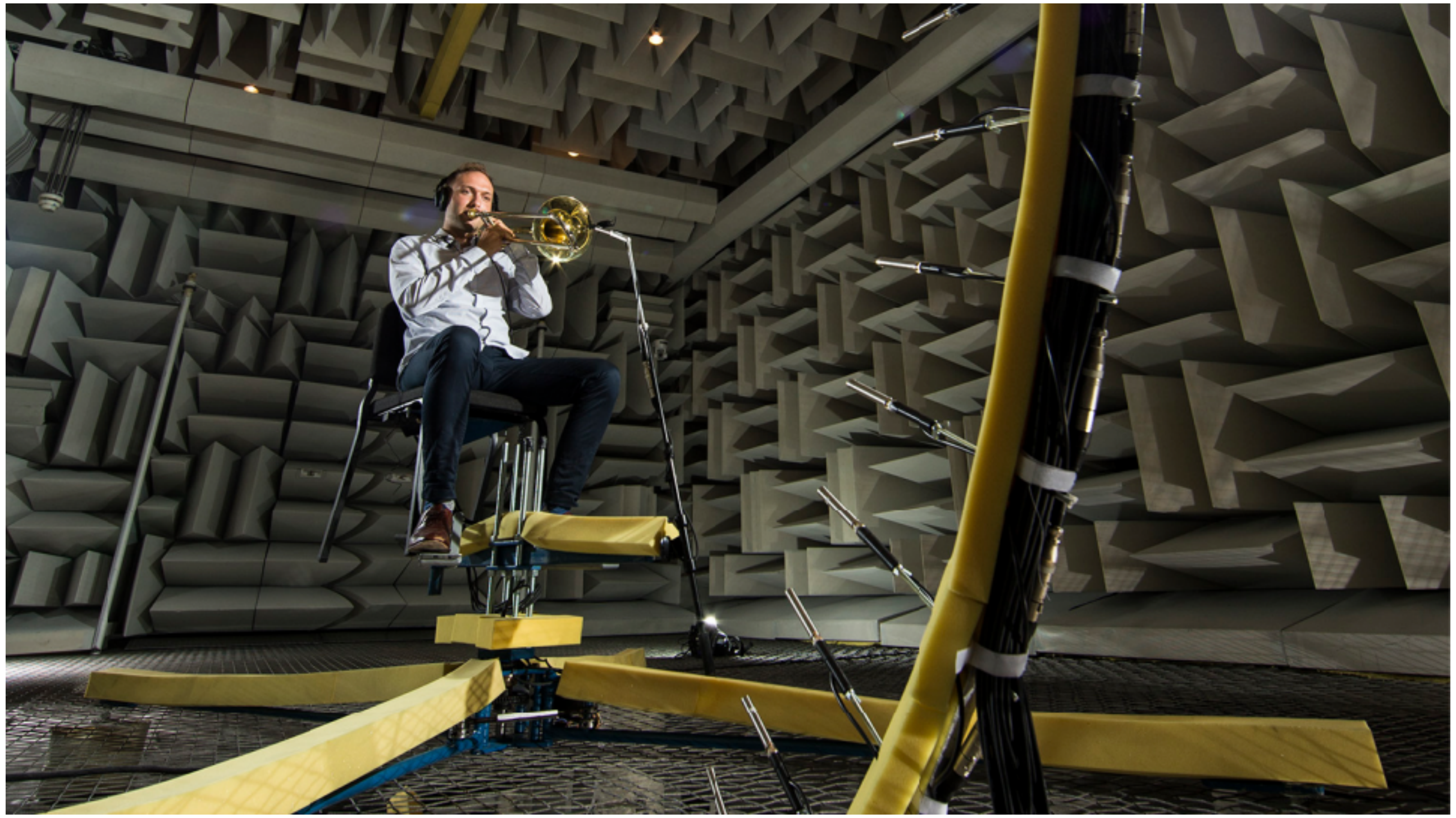
1,680 Retweets **1,634** Likes

24 1.7K 1.6K



$$(a + b)^3$$
$$= a^3 + 3a^2b + 3ab^2 + b^3$$





```
function processEntriesImperative (entries) {
    const csvCopy = entries.slice()

    csvCopy.sort(function (a, b) {
        if (a['Date Created'] === b['Date Created']) return 0
        if (a['Date Created'] > b['Date Created']) return -1
        if (a['Date Created'] < b['Date Created']) return 1
    })

    const seenAlready = {}

    const finalArray = []

    for (let i = 0; i < csvCopy.length; i++) {
        if (!seenAlready[csvCopy[i]['Your Name']]) {
            seenAlready[csvCopy[i]['Your Name']] = true
            finalArray.push(csvCopy[i])
        }
    }

    return finalArray
}
```

```
const R = require('ramda')

const processEntriesFunctional = R.pipe(
    R.sort(R.descend(R.prop('Date Created'))),
    R.uniqBy(R.prop('Your Name'))
)
```

Add numbers from 1 to n (exclusive?)

reminder...

O(n): sum = 1 + 2 + 3 + … + (n - 1)

O(1): sum = n * (n - 1) / 2

(we'll do the naive way, for the sake of demonstration)

like JS `reduce`

list from 1 to

```
series n =  foldl (+) 0 [1..(n-1)]
```

Function definition

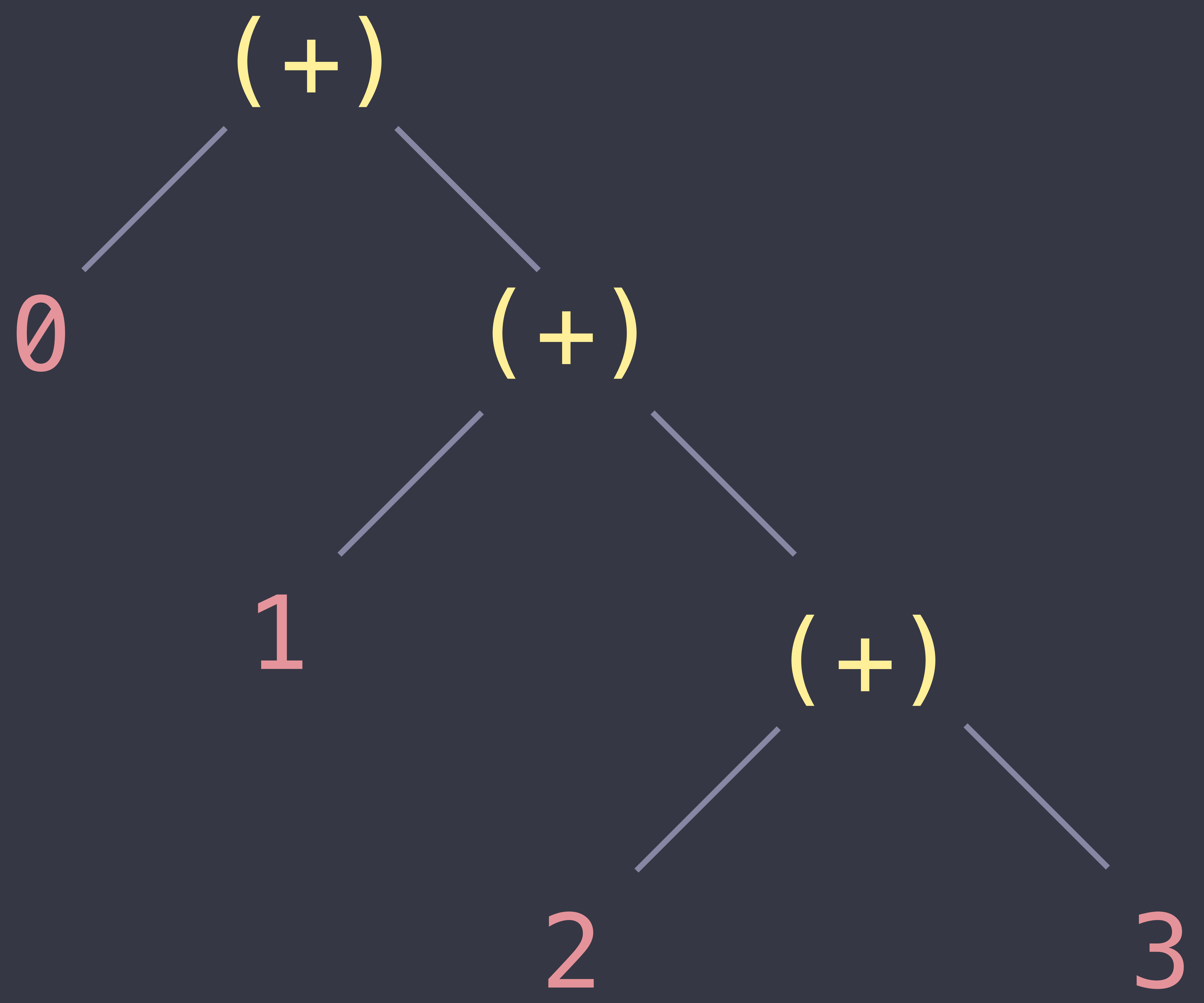
`add` function

start value

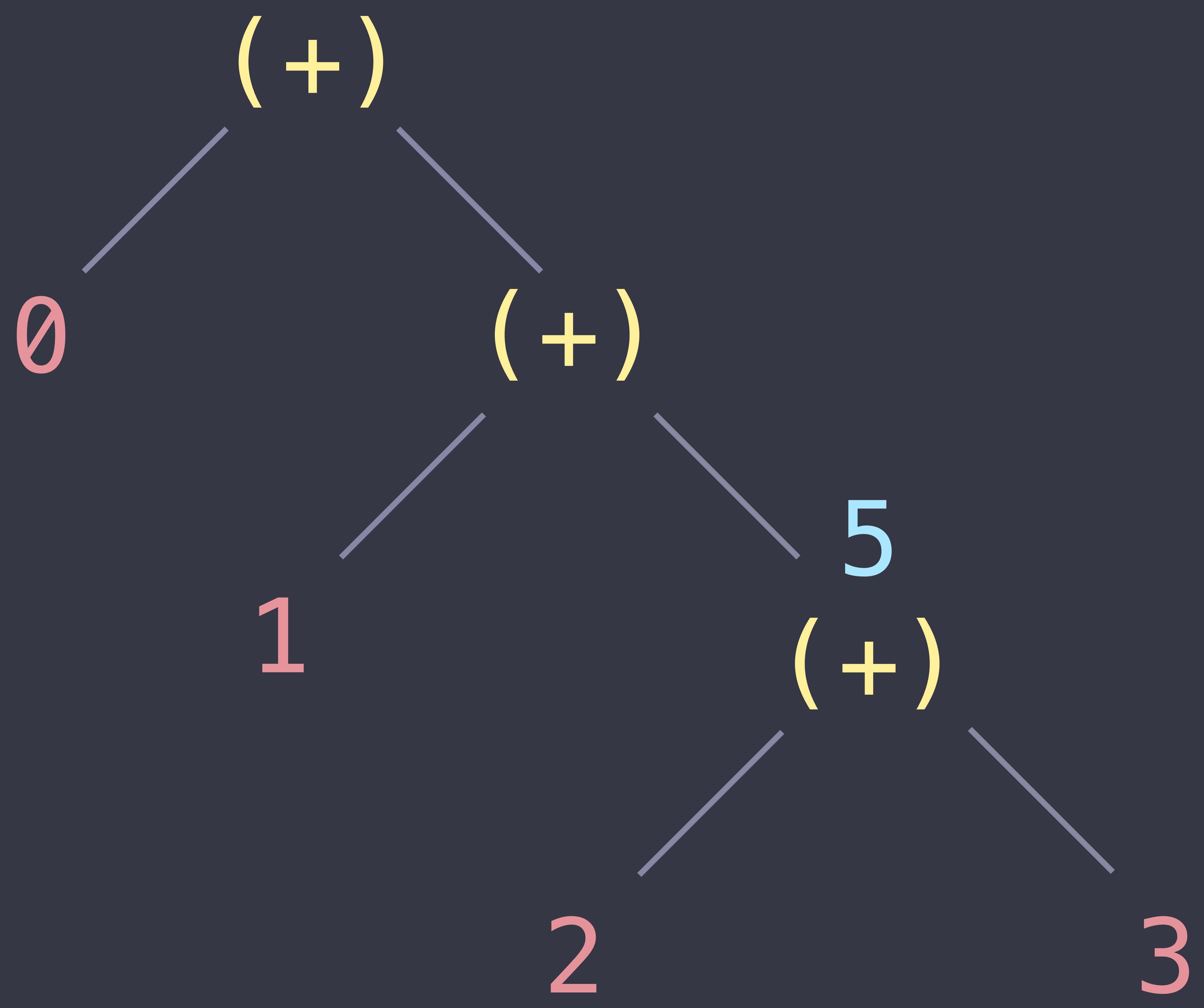
```
series 4 = foldl (+) 0 [1..(4-1)]
```

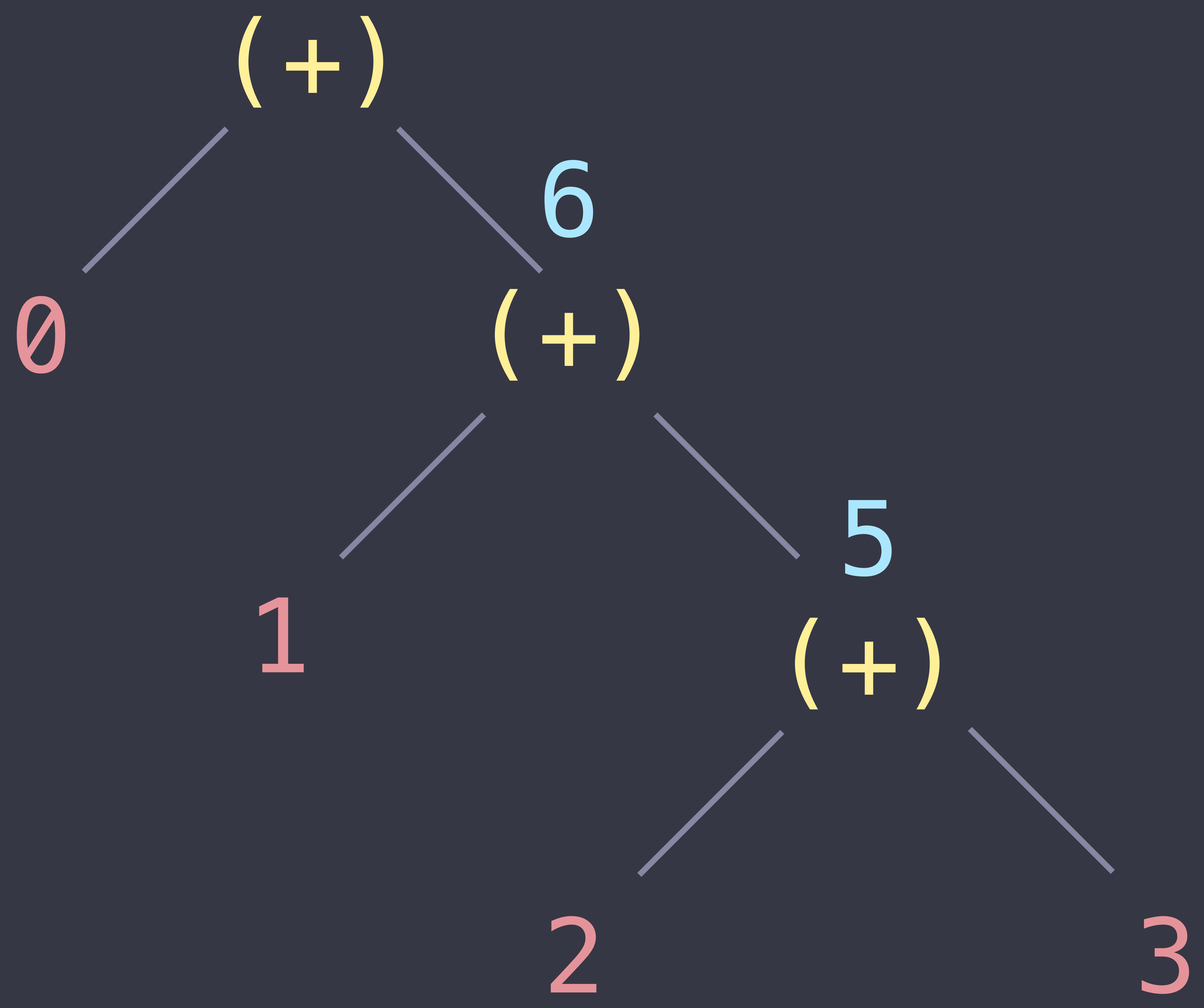
```haskell
series 4 = foldl (+) 0 [1, 2, 3]
```

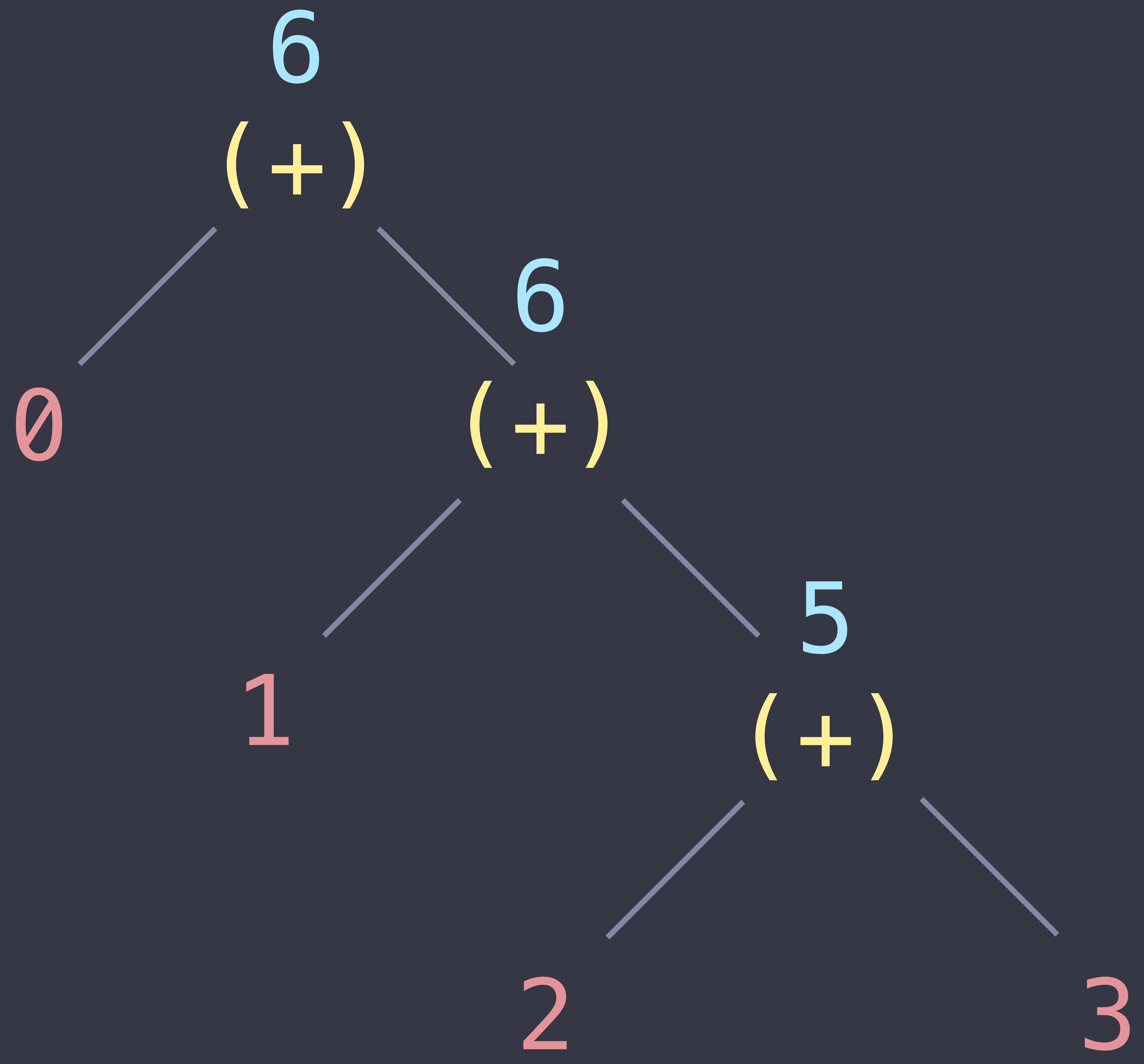
```
foldl (+) 0 [1, 2, 3]
```



(+)
0
(+)
1
(+)
2
3


(+)
0
(+)
1
5
(+)
2
3


(+)
0
6
(+)
1
5
(+)
2
3


6
(+)
0
6
(+)
1
5
(+)
2
3

6

Case Study: Mergesort

- Split list in half
- Recursively sort each half
- Merge sorted halves into sorted list
  - Take smaller of the two leading elements
  - Keep doing that until nothing left to take

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

Merge sorting empty list = empty list

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
              where (left, right) = splitAt midpoint xs
                    midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                    then x : merge xs (y:ys)
                    else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

Merge sorting singleton = singleton

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

Merge sorting anything else =
`merge` sorted `left` with sorted `right`

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
  where (left, right) = splitAt midpoint xs
        midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

Any other list

Recursion

`left` and `right` are results of splitting at `midpoint`

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

Two return values, in a tuple

Built-in, but not hard to define

and `midpoint` is the length / 2, rounded down

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
              where (left, right) = splitAt midpoint xs
                    midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

merging an empty list with 2nd list = 2nd list

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

merging 1st list with an empty list = 1st list

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
              where (left, right) = splitAt midpoint xs
                    midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                    then x : merge xs (y:ys)
                    else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

merging two lists, each starting w/ some val...

list beginning with some x

list beginning with some y

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

is the smaller val concat'd to merged remainder

construct list beginning with x

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

is the smaller val concat'd to merged remainder

construct list beginning with y

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

sorted = mergesorting this particular list

so what?

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
              where (left, right) = splitAt midpoint xs
                    midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

many function applications

& syntactic sugar for function applications

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1]
```

expressions evaluate to produce values

no such thing as instructions which cause effects

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
  where (left, right) = splitAt midpoint xs
        midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1]
```

defining nouns / relationships, not linear procedures

```
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1]
```

no mutation of state anywhere - all constant

much less specification of order - compiler handles it

```
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1]
```

so we can take this...

```haskell
mergesort []  = []
mergesort [x] = [x]
mergesort xs  = merge (mergesort left) (mergesort right)
                where (left, right) = splitAt midpoint xs
                      midpoint      = length xs `quot` 2

merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

...and change it to this...

```haskell
mergesort [x] = [x]
mergesort []  = []
mergesort xs  = merge (mergesort left) (mergesort right)
              where midpoint      = length xs `quot` 2
                    (left, right) = splitAt midpoint xs

merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys
merge []     ys     = ys
merge xs     []     = xs

sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

```haskell
sorted = mergesort [4, 2, 6, 9, 1] -- [1, 2, 4, 6, 9]
```

...or even this. Still works!

```haskell
merge (x:xs) (y:ys) = if x <= y
                      then x : merge xs (y:ys)
                      else y : merge (x:xs) ys
merge []     ys     = ys
merge xs     []     = xs

mergesort [x] = [x]
mergesort []  = []
mergesort xs  = merge (mergesort left) (mergesort right)
                where midpoint      = length xs `quot` 2
                      (left, right) = splitAt midpoint xs
```

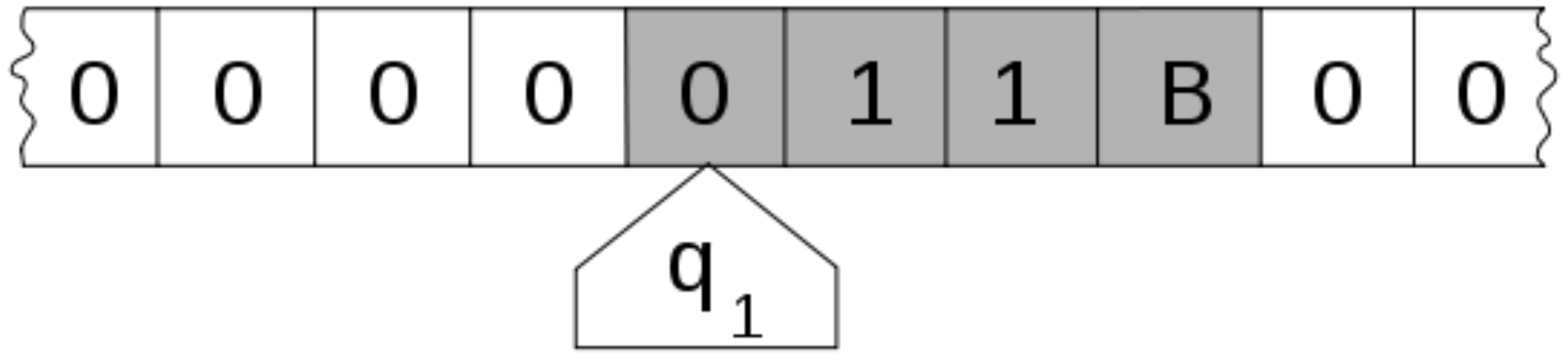

HISTORY
λ

Theories of Computability

Alonzo Church   Alan Turing

*(*both benefitted from many other mathematicians, including Gödel, Haskell, Schönfinkel, Frege, Rósza Péter etc.)*

$$(\lambda xy.x\ y\ ((\lambda fab.fba)\ y))$$



- ca. 1928 develops Lambda Calculus
- all computation can be expressed as applications of pure functions

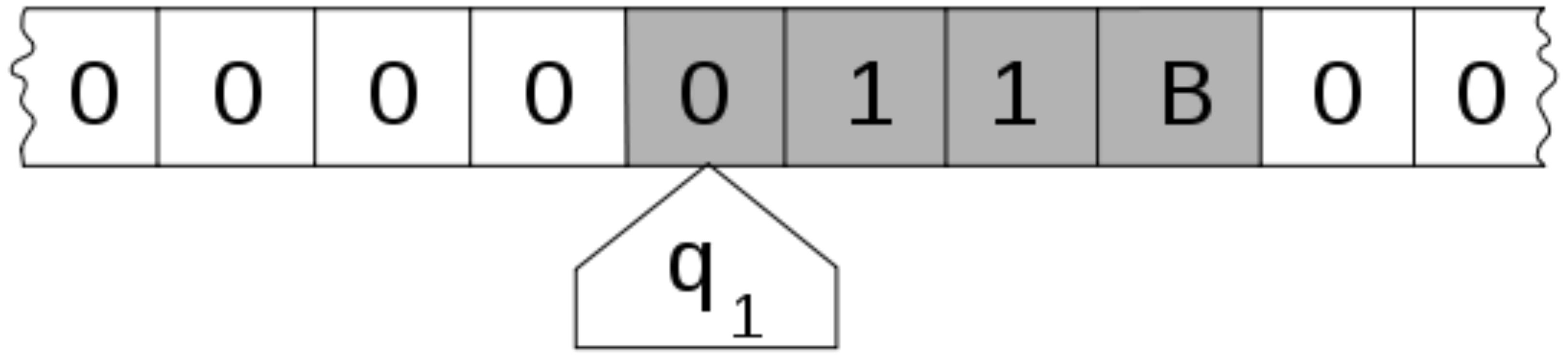
- ca. 1936 develops Turing Machine
- all computation can be expressed as state machine changes

Church-Turing Equivalence

Two ways of expressing the same concept.

Everything one can do, the other can too.

$$(\lambda xy.x\ y\ ((\lambda fab.fba)\ y))$$



exciting because it means code can be *entirely abstract*

exciting because it means we can make *real computers*