Programação Funcional para Amazonas e Cavaleiros


Pela honra, pela glória e pelo λ!

Programação funcional em férias nos anos 1930

OO? More like NO NO!

"… since modularity is the key to successful programming, functional programming offers important advantages for software development."

–John Hughes, "Why Functional Programming Matters"

Dividir para conquistar

```
pipe(map(mappingFn), reduce(reducingFn))(bigData)
```

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

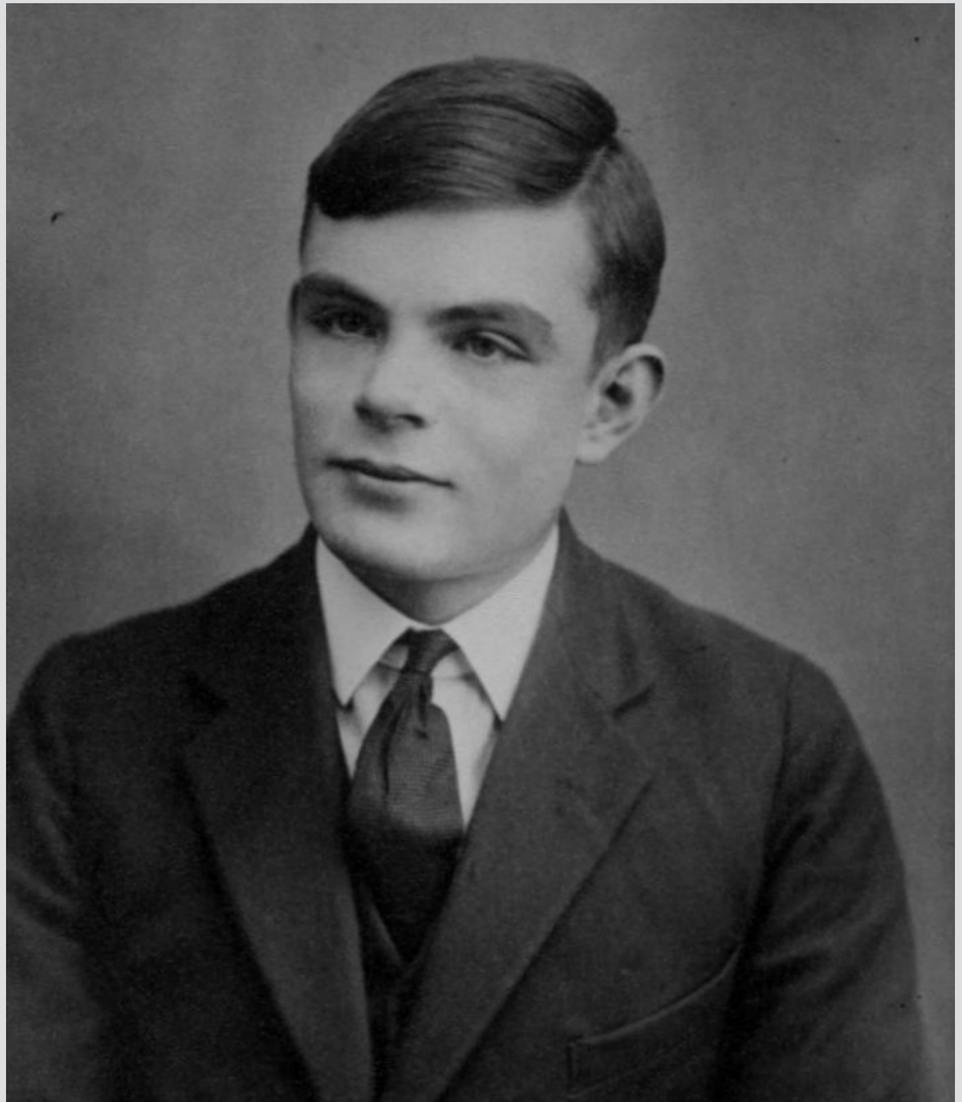
Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with

Para as coisas mais complicadas,
precisamos da *Matemática*

Cálculo Lambda

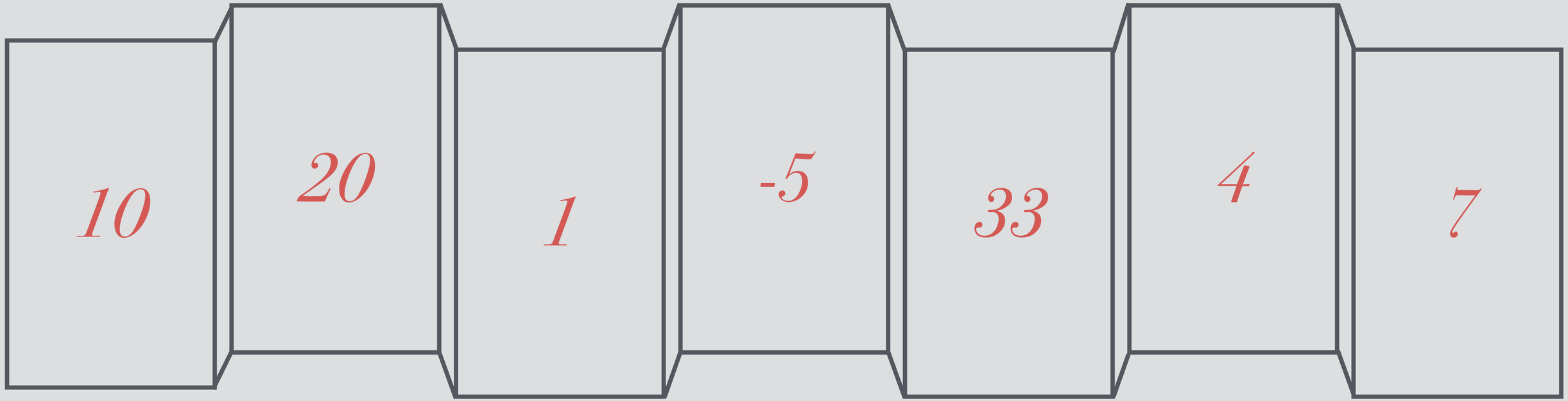


“Philosophers might argue as to whether mathematical systems are ‘discovered’ or ‘devised’, but the same system arising in two different contexts argues that here the correct word is ‘discovered’.”

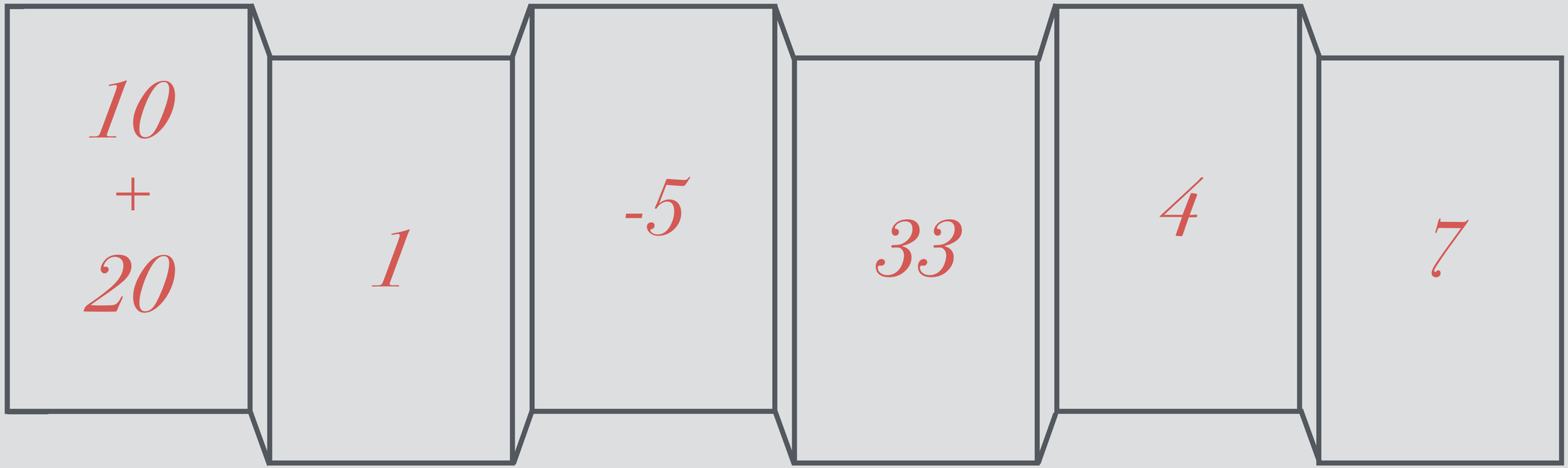
–Philip Wadler, “Propositions as Types”

```
function reduce(fn, acc, array) {
  if (array.length === 0) {
    return acc;
  } else {
    return reduce(fn, fn(acc, first(array)), rest(array));
  }
}
```

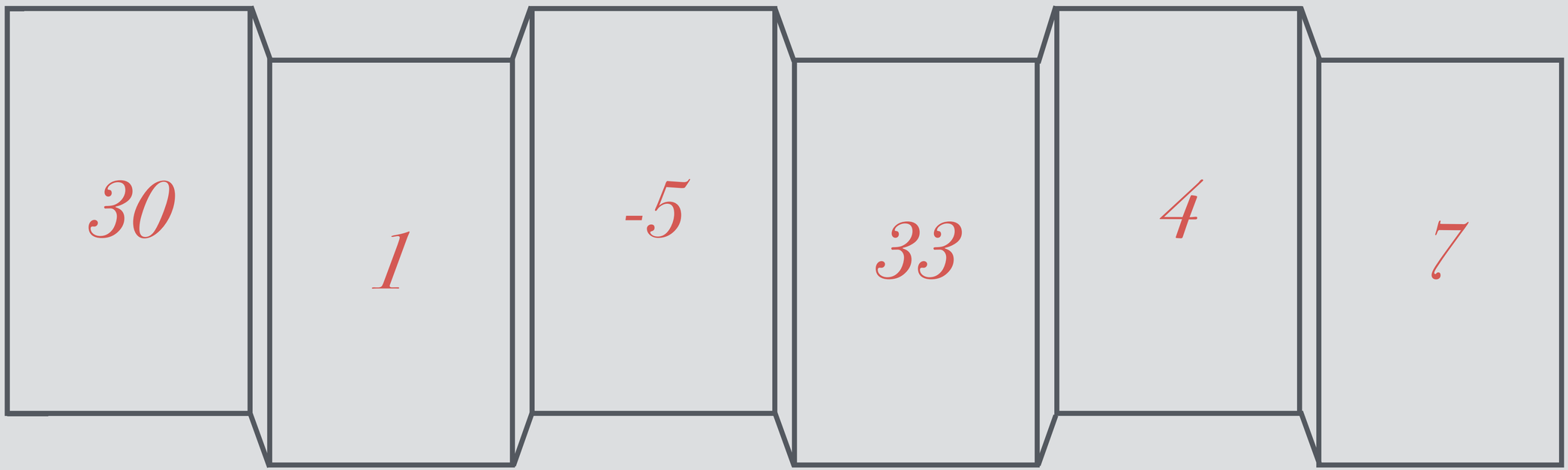
```
function foldLeft(fn, acc, array) {
  if (array.length === 0) {
    return acc;
  } else {
    return foldLeft(fn, fn(acc, first(array)), rest(array));
  }
}
```



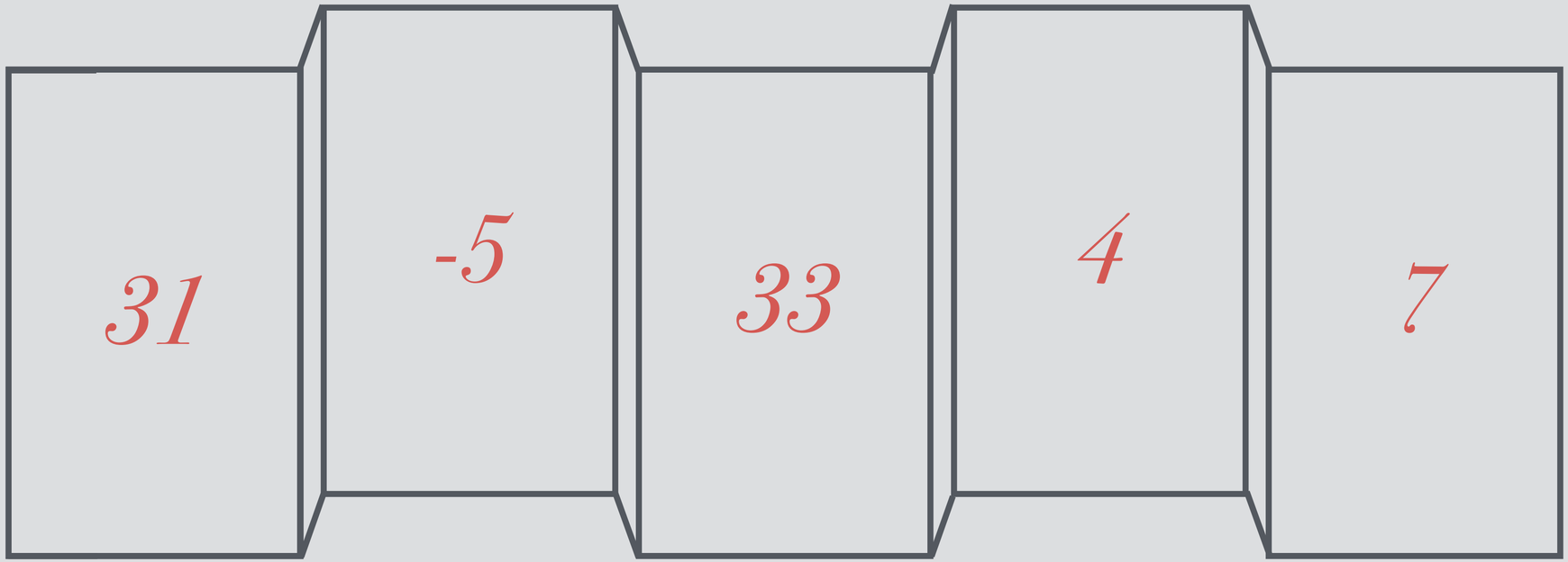

```
foldLeft((sum, n) ⇒ sum + n, 0, […])
```



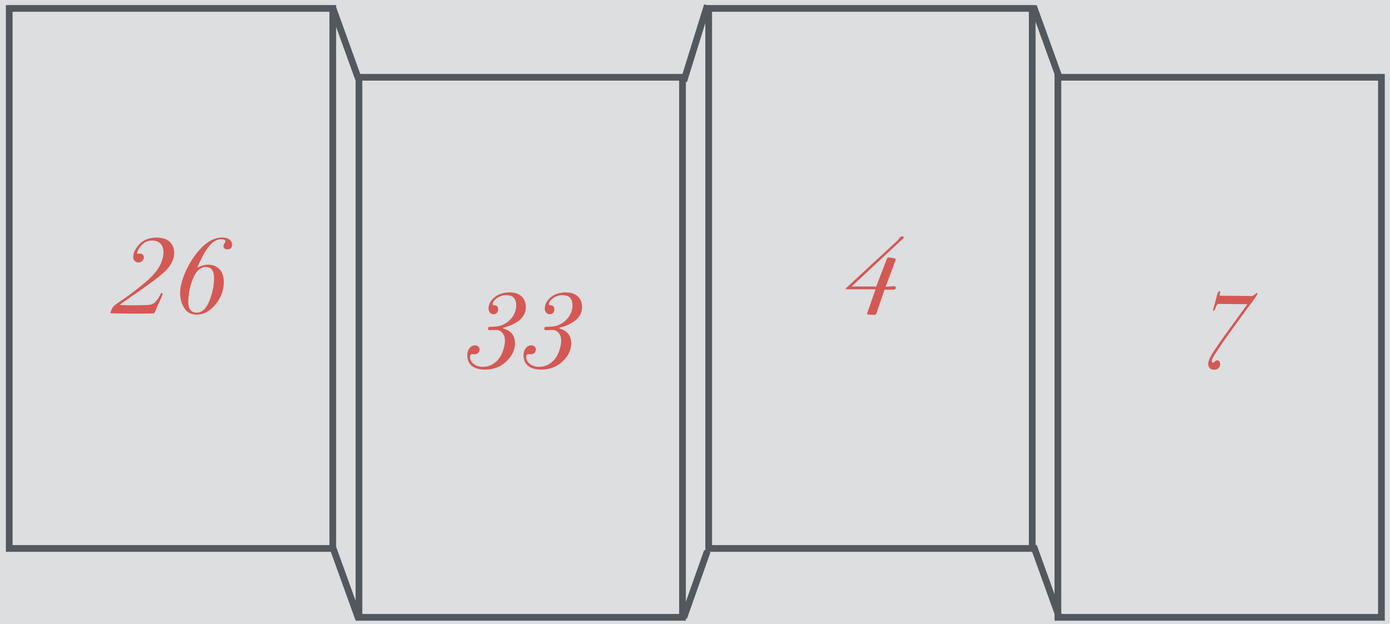

```
foldLeft((sum, n) ⇒ sum + n, 0, […])
```



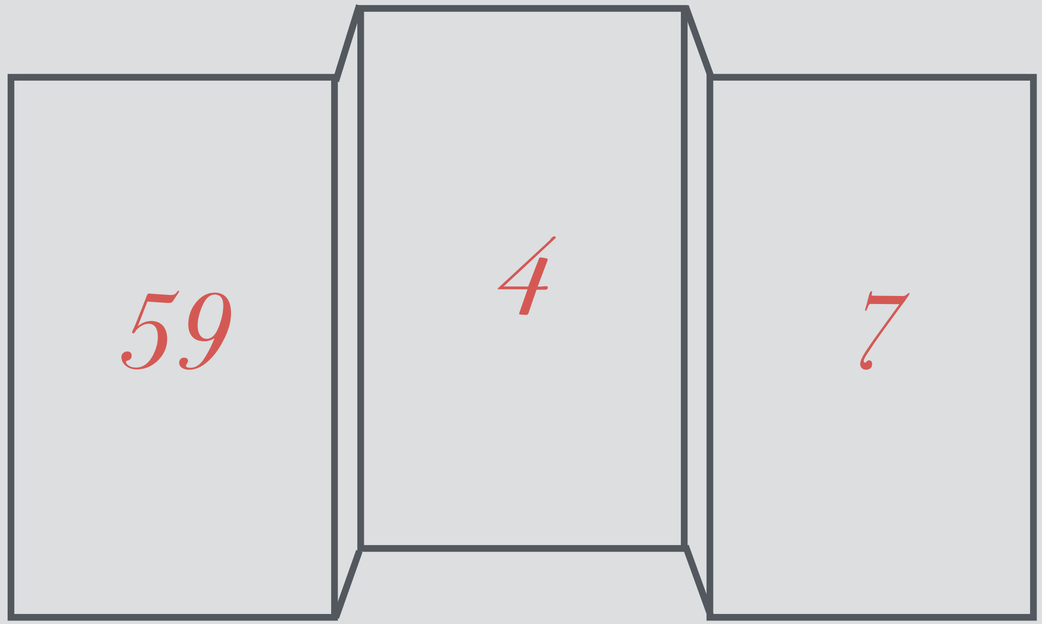

```
foldLeft((sum, n) ⇒ sum + n, 0, […])
```



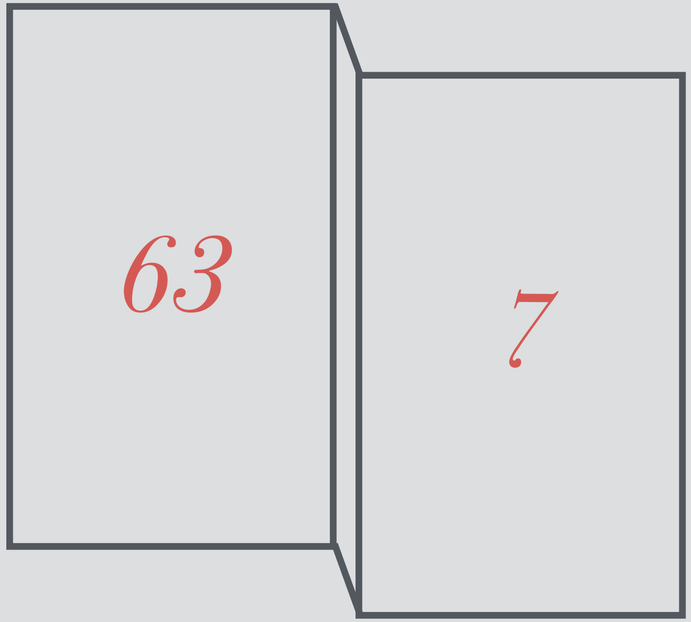

```
foldLeft((sum, n) ⇒ sum + n, 0, […])
```




```
foldLeft((sum, n) ⇒ sum + n, 0, […])
```




```
foldLeft((sum, n) ⇒ sum + n, 0, […])
```




```
foldLeft((sum, n) ⇒ sum + n, 0, […])
```


```
foldLeft((sum, n) ⇒ sum + n, 0, […])
```

map
filter
take
takeWhile
…

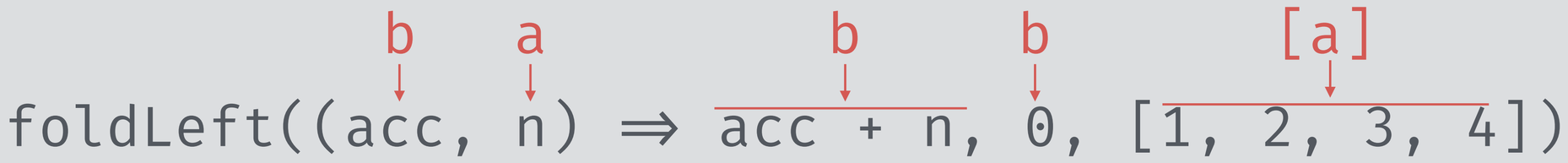
Currying

```
foldl ::(b → a → b) → b → [a] → b
```

```
foldl ::(b → a → b) → b → [a] → b
```

```
foldLeft((acc, n) ⇒ acc + n, 0, [1, 2, 3, 4])
```

```
foldl ::(b → a → b) → b → [a] → b
```



```
Prelude> let mul xs = foldl (\m n → m * n) 1 xs
Prelude> mul [1, 2, 3, 4]
24
```

```
Prelude> let mul = foldl (*) 1
Prelude> mul [1, 2, 3, 4]
24
```

```
function curry(fn) {
  return (a) ⇒ fn(a);
}

function curry2(fn) {
  return (first) ⇒ (last) ⇒ fn(first, last);
}

function curry3(fn) {
  return (first) ⇒ (middle) ⇒ (last) ⇒ fn(first, middle, last);
}
```

```
> let mul = curry3(foldLeft)((a, b) ⇒ a * b)(1);
> mul([1, 2, 3, 4]);
24
```

Pipelines

```
let values = [1, 2, 3, 4, 5, 6];
values
   .map((n) ⇒ n * 10)
   .filter((n) ⇒ n > 30)
   .reduce((a, b) ⇒ a * b, 1);
```

```
let whoVotedForWhom = { John: "Alice", Bob: "Alice", Alice:
"John", Julia: "Bob" };

_.chain(whoVotedForWhom)
  .pairs()
  .reduce((votes, [voter, votee]) => {
    return _.extend(votes,
                    { [votee]: (votes[votee] || [])
                                 .concat([voter]) });
  }, {})
  .value();
```

```
=> { Alice: [ 'John', 'Bob' ], John: [ 'Alice' ], Bob: [ 'Julia' ] }
```

```
R.pipe(R.toPairs,
                  R.reduce((votes, [voter, votee]) ⇒ {
                    return R.merge(votes,
                       { [votee]: (votes[votee] || [])
                                    .concat([voter]) });
                  }, {}))(whoVotedForWhom);
```

Vamos trabalhar