Functional Programming Style

→ Scala influenced ⊕ Distributed Computing

→ Map → Shuffle → Reduce

↓ Functional Programming

① Comes from algebra (math)

$f(x) = x + 3$

$f(5) = 8$

⚠ Functions have a single output!

② Output of a function just depends on input

↳ "Stateless functions"

⚠ Data state changing avoided

→ Concepts:

① First class and Higher order functions:

↳ Functions that can take (as argument) & return other functions

② Pure Functions: have no side-effects (in memory or I/O)

- if the result of a pure expression is not used, it can be removed safely
- if a pure function is called with arguments with no side effects, the result will be constant → possible called "memoization"
- if there is no data dependency between two functions, their order can be reversed or run in parallel
- The language does not allow side-effects → compiler can rearrange expressions to improve efficiency.

③ Recursion: iteration can usually be achieved by recursivity

④ Strict vs non-strict evaluation: strict (eager) evaluation
non-strict (lazy) evaluation

⑤ Type systems

⑥ Referential Transparency

$x = x + 10$ → NO REFERENTIAL TRANSPARENCY

int plusone (int x) {return x+1;} → REF. TRANSPARENCY

Procedural / Imperative Programming

for i in range (1, x):
    do_something()

✗ ⚠ Not suitable for distributed programming

ⓘ Data Structures different implementation as an "implicit" counterpart.

Example Functional Programming

```
const numbers = [1, 2, 3, 4, 5]

const result = numbers.filter(lambda x: x%2!=0).reduce(lambda a, b: a + b*10)
```

Pure Functions Analogy:

- Functions should be "encapsulated", just affecting things in their scope.
- Not affecting input values (no modification)
  - → Making a copy of its input data

The Spark DAG:

Idea → If we where to copy data each time before it is passed to a function → Rapidly run out of memory!!

↓

Lazy Evaluation: Before executing, Spark constructs a "DAG" (Direct Acilic Graph) of calculations to be held → Check if it can wait till the moment to get the data