# CS2030S Recitation

## Problem Set 8

brian

2026-04-09

National University of Singapore

# Recap: Functors

# Recap: Functors

- A functor is a *structure* with 2 methods (of, map) obeying two laws:
  1. Identity morphism (mapping identity function gives you the same functor)
     - ∀ functor: functor.map(x → x) ≡ functor
  2. Composition morphism (For ever 2 maps, there's a map that composes the 2)
     - ∀ functor, f,g: functor.map(x → f(x)).map(y → g(y)) ≡ functor.map(x → g(f(x)))

# Recap: Monads

# Recap: Monads

- Simple definition
- A Monad in X is a monoid in the category of endofunctors of X

# Question 1a: Background

```
1  class Monad<T> {
2    private T x;
3    private Monad(T x) { this.x = x; }
4
5    public static <T> Monad<T> of (T x) { return new Monad<>(x); }
6
7    public T get() { return x; }
8
9    public <R> Monad<R> flatMap(Transformer<? super T,
10       ? extends Monad<? extends R>> f) {
11     return new Monad<>(f.transform(this.x).get());
12   }
13
14   public <R> Monad<R> map(Transformer<? super T, ? extends R> f) {
15     return this.flatmap(???); // <-- What to do here?
16   }
17 }
```

# Recap: Monads

- A box with context (side information)
- Has a `flatMap` method (takes in a function f: `X` → `Monad<X>`, composes contexts)
- Has a `of` method (Wrap `X` into the `Monad<X>`)
- Has to fulfill 3 laws
  1. Left identity (`of` should not add effects/context)
     - ∀ x, f: Monad.of(x).flatMap(y → f(y)) ≡ f(x)
  2. Right identity (Composing the effects of `of` should not change the context)
     - ∀ m: m.flatMap(x → Monad.of(x)) ≡ monad
  3. Associative
     - ∀ m, f, g: m.flatMap(x → f(x)).flatMap(y → g(y))
       ≡ m.flatMap(x → f(x).flatMap(y → g(y)))

# Question 1a

```
1  class Monad<T> {
2    private T x;
3    private Monad(T x) { this.x = x; }
4
5    public static <T> Monad<T> of (T x) { return new Monad<>(x); }
6
7    public T get() { return x; }
8
9    public <R> Monad<R> flatMap(Transformer<? super T,
10       ? extends Monad<? extends R>> f) {
11     return new Monad<>(f.transform(this.x).get());
12   }
13
14   public <R> Monad<R> map(Transformer<? super T, ? extends R> f) {
15     return this.flatmap(???); // <-- What to do here?
16   }
17 }
```

# Question 1a

# Question 1a

```
1  class Monad<T> {
2    private T x;
3    private Monad(T x) { this.x = x; }
4
5    public static <T> Monad<T> of (T x) { return new Monad<>(x); }
6
7    public T get() { return x; }
8
9    public <R> Monad<R> flatMap(Transformer<? super T,
10       ? extends Monad<? extends R>> f) {
11     return new Monad<>(f.transform(this.x).get());
12   }
13
14   public <R> Monad<R> map(Transformer<? super T, ? extends R> f) {
15     return this.flatmap(x -> Monad.of(f.transform(x)))
16   }
17 }
```

# Question 1b

Show that the monad preserves composition

- Show that
  - `m.map(x → f(x)).map(y → g(y)) ≡ m.map(x → g(f(x)))`

# Question 1b

m.map(x → f(x)).map(y → g(y)) (Starting)

m.flatMap(x → Monad.of(f(x))).flatMap(y → Monad.of(g(y))) (by β-reduction/substitution)

m.flatMap(x → Monad.of(f(x)).flatMap(y → Monad.of(g(y)))) (by Associativity)

m.flatMap(x → Monad.of(g(f(x)))) (by Left Identity)

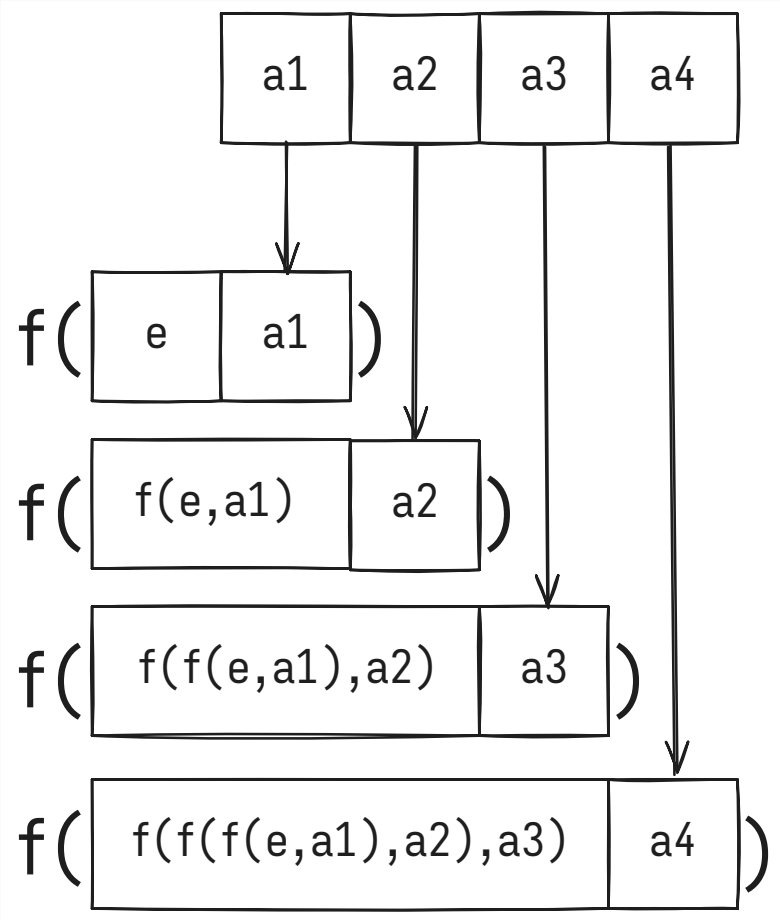
m.map(x → g(f(x))) (by Implementation)

# Recap: Concurrency

- Sequential
  - Do things in order on one thread
- Concurrent
  - One thread does multiple things by context switching very quickly
- Parallel
  - Actually do multiple things at the same time

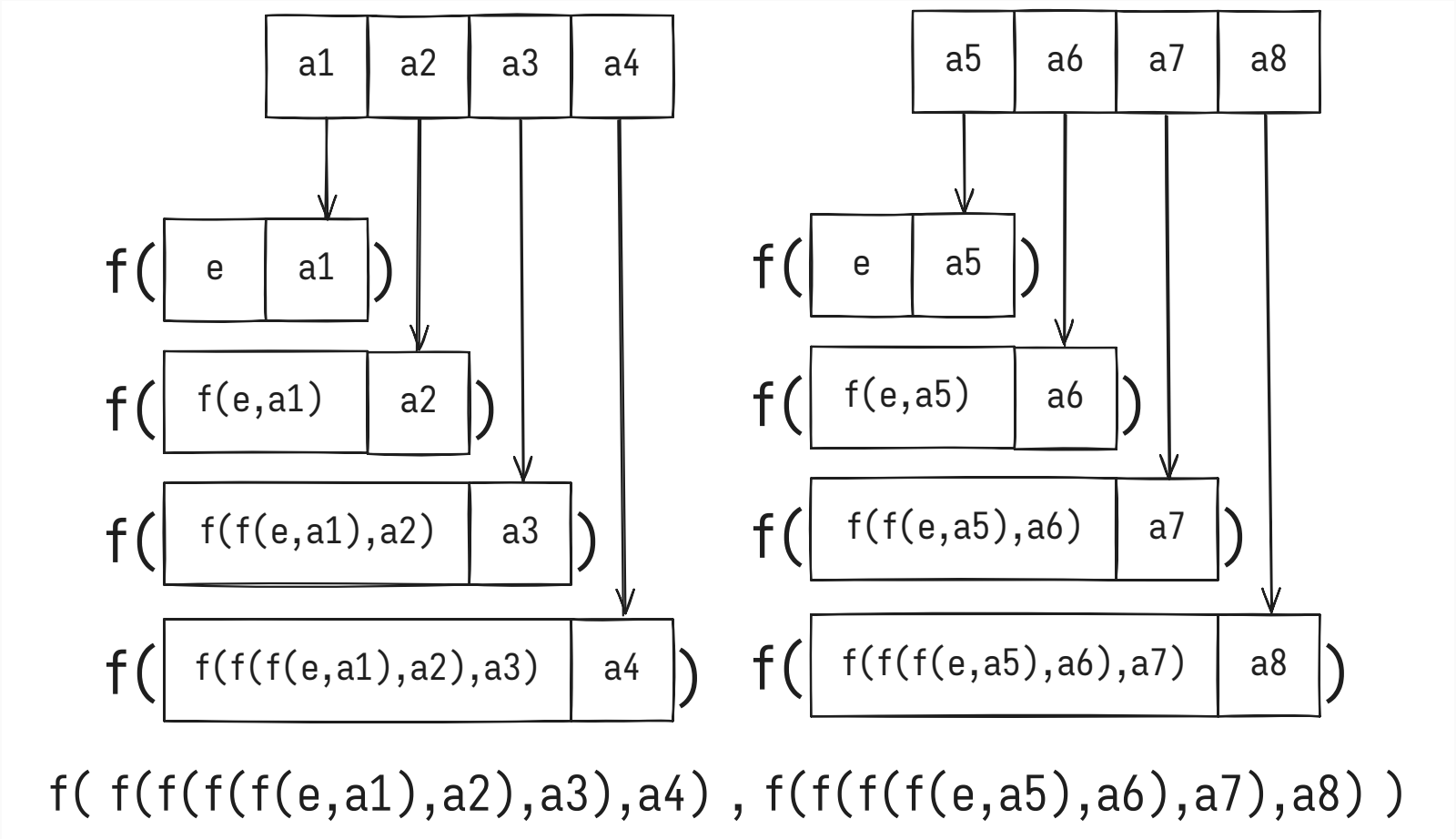
# Recap: Reduce Sequential

```
T reduce(T e, BinaryOperator<T> f)
```



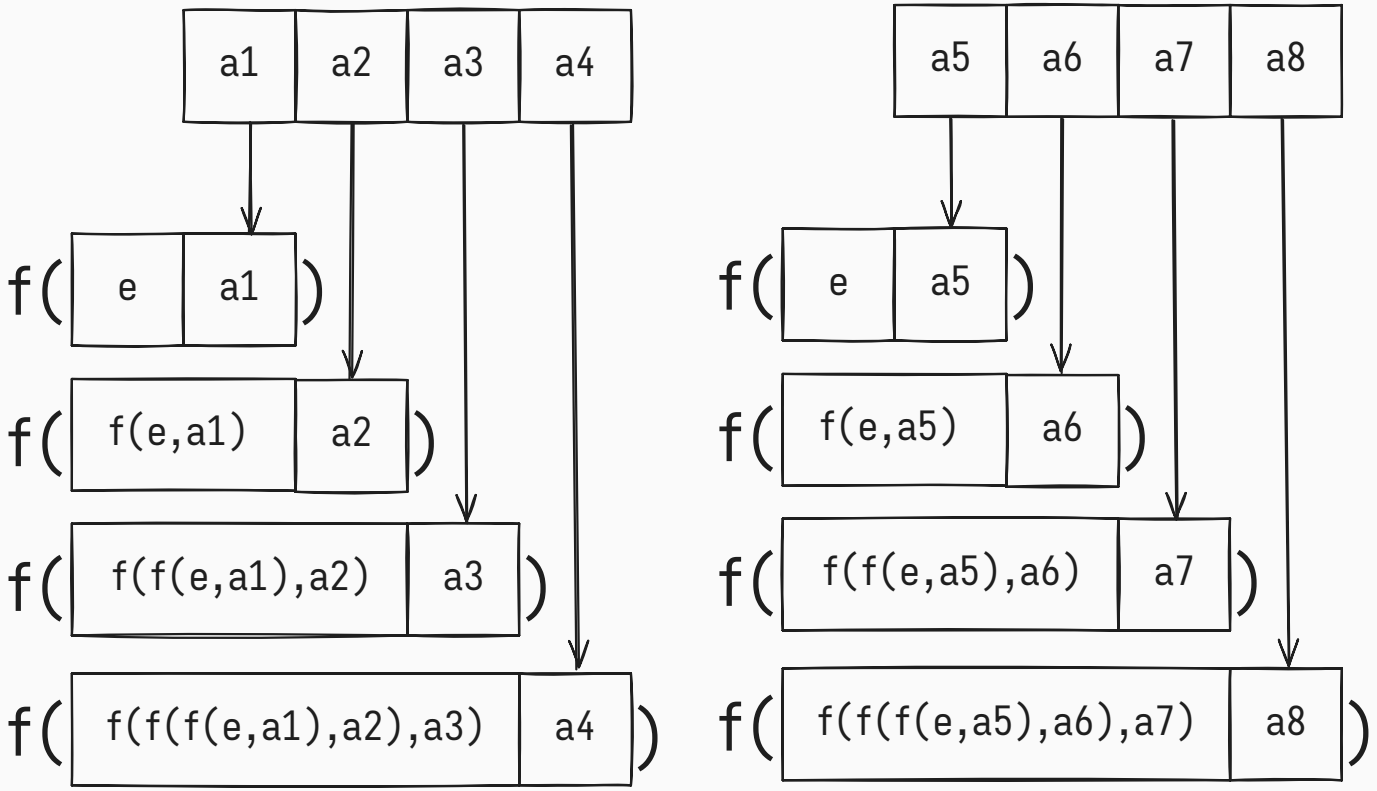
# Recap: Reduce Parallel

```
T reduce(T e, BinaryOperator<T> f)
```



# Recap: Reduce Parallel

```
<U> U reduce(U e, BiFunction<U,? super T,U> f, BinaryOperator<U> g)
```




g( f(f(f(f(e,a1),a2),a3),a4) , f(f(f(f(e,a5),a6),a7),a8) )

# Question 2a and 2b

What is the returned value for each of the pipelines

1.
```
1 Stream.of(1,2,3,4)
2       .reduce(0, (a, x) -> (2 * a) + x, (a1, a2) -> a1 + a2);
```

2.
```
1 Stream.of(1,2,3,4)
2       .parallel()
3       .reduce(0, (a, x) -> (2 * a) + x, (a1, a2) -> a1 + a2);
```

# Question 2a and 2b

So why is it so different?

- brian shows on whiteboard execution
- Accumulator is not associative $f(f(a,b),c) \neq f(a,f(b,c))$
- brian shows why not associative on whiteboard again
- Combiner and accumulator are also not compatible $\mathrm{comb}(u, \mathrm{acc}(\mathrm{id}, t)) \neq \mathrm{acc}(u, t)$

# Question 3

Recall `estimatePi` from Lab 0

```
1  public static double estimatePi(int numOfPoints, int seed) {
2    RandomPoint.setSeed(seed);
3    Circle c = new Circle(new Point(0.5, 0.5), 0.5);
4    int n = 0;
5    for (int i = 0; i < numOfPoints; i++) {
6      Point p = new RandomPoint(0, 1, 0, 1);
7      if (c.contains(p)) {
8        System.out.println(p);
9        n++;
10     }
11   }
12   return 4.0 * n / numOfPoints;
13 }
```

What can we say about the probability of a random point being in the circle?

# Question 3

# Question 3

- Does parallelization make it faster?
  - No it's costly to make new threads
  - Just creating objects is fast
  - Parallelization is good if each task is very costly (in this case it's not)

## Why is the number different

- Each thread has access to the same random seed
- The order which the threads interleave is random
- When limit happens, the threads may have produced more than necessary elements and would be chopped fulfill
- So what's left is a stream that can have different random points each run due to the randomness
- anyway doesn't matter don't think too much

# The End

bye!

Download slides here: