Week 10 Lab Session

CS2030S AY21/22 Semester 2

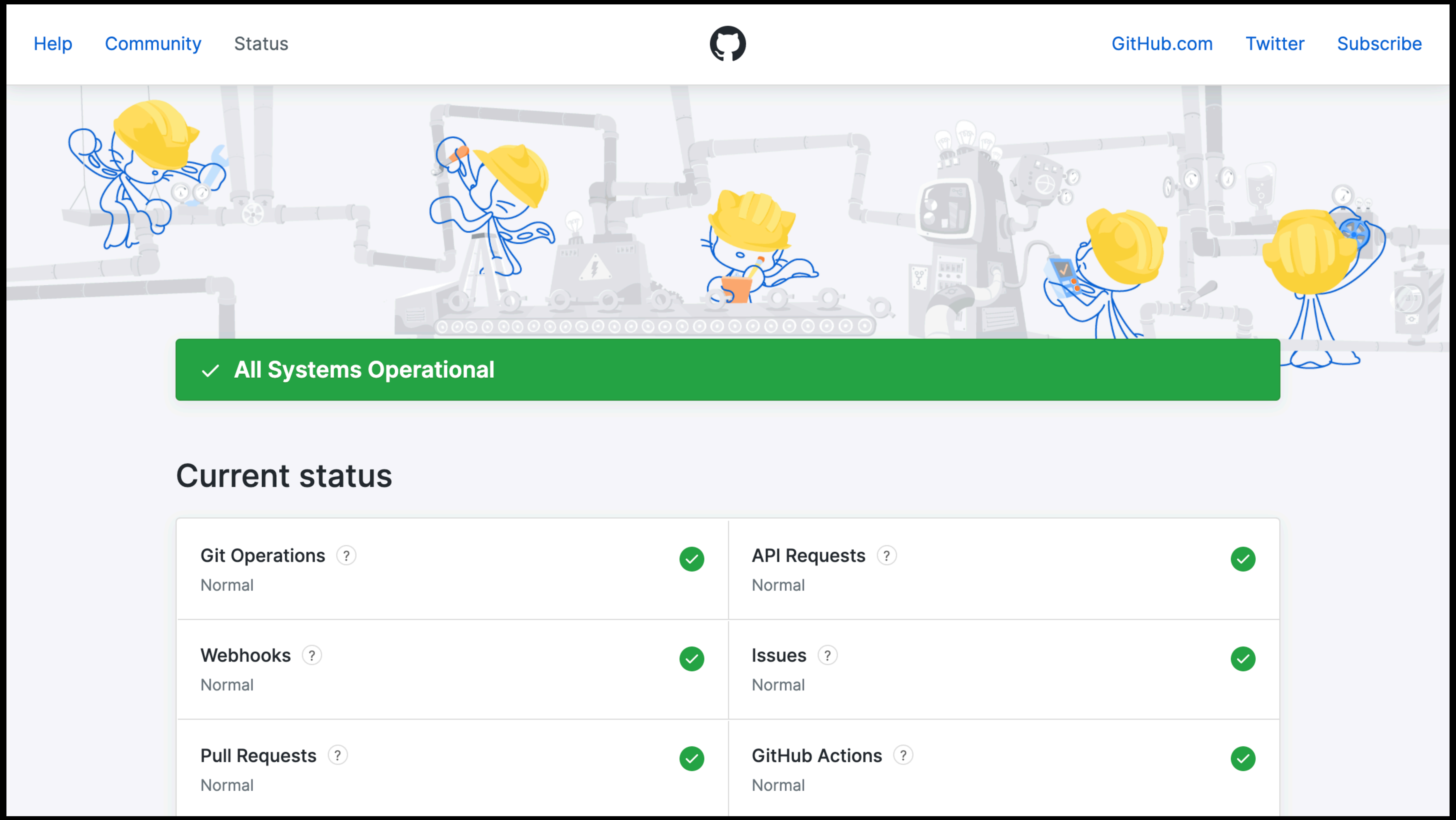
Lab 14B

Yan Xiaozhi (David)
@david_eom
yan_xiaozhi@u.nus.edu

24 Mar 2022

Admin

- Contact tracing & QR code
- Lab 6 marking is out
    - Do approach me if you have any questions
- Submission and GitHub downtime
- Lab schedule for next two weeks:
    - Next week: mock PE2
    - The week after: lab 7 debrief



✓ All Systems Operational

Current status

Git Operations ? ✓ Normal	API Requests ? ✓ Normal
Webhooks ? ✓ Normal	Issues ? ✓ Normal
Pull Requests ? ✓ Normal	GitHub Actions ? ✓ Normal

Lab 6 Feedback

Grading Scheme

- 10 marks correctness, 2 marks documentation
- -1 / -0.5 for each bug
- -1 for each raw type and abuse of `@SuppressWarnings`
- -1 for each missing PECS
- -1 / -2 for missing implementation of various methods
- Up to -2 for egregious styling breach

A Quick Revisit

- Design `Lazy<T>` class that contains a `Maybe` and a `Producer`
- Factory methods `of(T v)` and `of(Producer<T> s)`
- `get()` with memoisation
- `toString()`, ? for values that have yet been computed
- `map`, `flatMap`, `filter` and `combine`
- `equals`
- No explicit checking
- PECS correctly applied for all methods

Lazy<T>::get

-
  ```
  public T get() {
    return this.producer.produce();
  }
  ```

Lazy<T>::get

-
  ```
  public T get() {
    return this.producer.produce();
  }
  ```

-
  ```
  public T get() {
    return this.value.orElseGet(this.producer);
  }
  ```

Lazy<T>::get

-
  ```
  public T get() {
    return this.producer.produce();
  }
  ```

-
  ```
  public T get() {
    return this.value.orElseGet(this.producer);
  }
  ```

-
  ```
  public T get() {
    T t = this.value.orElseGet(this.producer);
    this.value = Maybe.of(t);
    return t;
  }
  ```

Lazy<T>::get

-
  ```
  public T get() {
    return this.producer.produce();
  }
  ```

-
  ```
  public T get() {
    return this.value.orElseGet(this.producer);
  }
  ```

-
  ```
  public T get() {
    T t = this.value.orElseGet(this.producer);
    this.value = Maybe.some(t);
    return t;
  }
  ```

Lazy<T>::toString

-
  ```
  public String toString() {
    return this.value.equals(Maybe.none())
      ? "?"
      : this.value.map(x -> String.valueOf(x));
  }
  ```

  - Might as well don’t use Maybe at all

-
  ```
  public String toString() {
    return this.value.map(String::valueOf).orElse(“?”);
  }
  ```

  - Much more readable, much more abstraction, chaining

`Lazy<T>` Other Methods

- Do NOT call get() directly, calls to get() must be delayed using a producer
- `map`: `new Lazy<R>(() -> tf.transform(this.get()))`
- `flatMap`: `new Lazy<R>(() -> tf.transform(this.get()).get())`
- `combine`: `new Lazy<R>(() -> f.combine(this.get(), s.get()))`
- `filter`: `new Lazy<Boolean>(() -> pred.test(this.get()))`

LazyList<T>::generate

-
  ```
  public static <T>  LazyList<T> generate(
      int n, T seed, Transformer<T, T> f) {
    LazyList<T> list = new LazyList<>(
      new ArrayList<Lazy<T>>());
    Lazy<T> curr = Lazy.of(seed);
    for (int i = 0; i < n; i++ ) {
      l.list.add(curr);
      curr = curr.map(x -> f.transform(x));
    }
    return list;
  }
  ```

- Thoughts on `Transformer<? super T, ? extends T>`?

LazyList<T>::indexOf

-
  ```
  public int indexOf(T v) {
    return this.list.indexOf(Lazy.of(v));
  }
  ```
- Make use of this abstraction!
  - Many re-implemented this method

Common Mistakes

- Not handling of `null` in `Lazy::equals`
    - i.e. `return this.get().equals(other.get())`
    - Good scripts use `Maybe`::`equals` instead after computation of value
        - `return this.value.equals(other.value)`
- Not making use of `List::indexOf`
    - Not using is fine, but some forgot to handle `null` or not use `equals`
- Missing `@Override` for `equals` and `toString`

Lab 7 Overview

Motivation

-
  ```
  class InfiniteList<T> {
    private Producer<T> head;
    private Producer<InfiniteList<T>> tail;
    …
  }
  ```

- Problems with the `InfiniteList` in lecture:

  - No memoisation, same values produced over and over again

  - Filtered values are `null`, cannot distinguish between a genuine `null` and a value that is not there

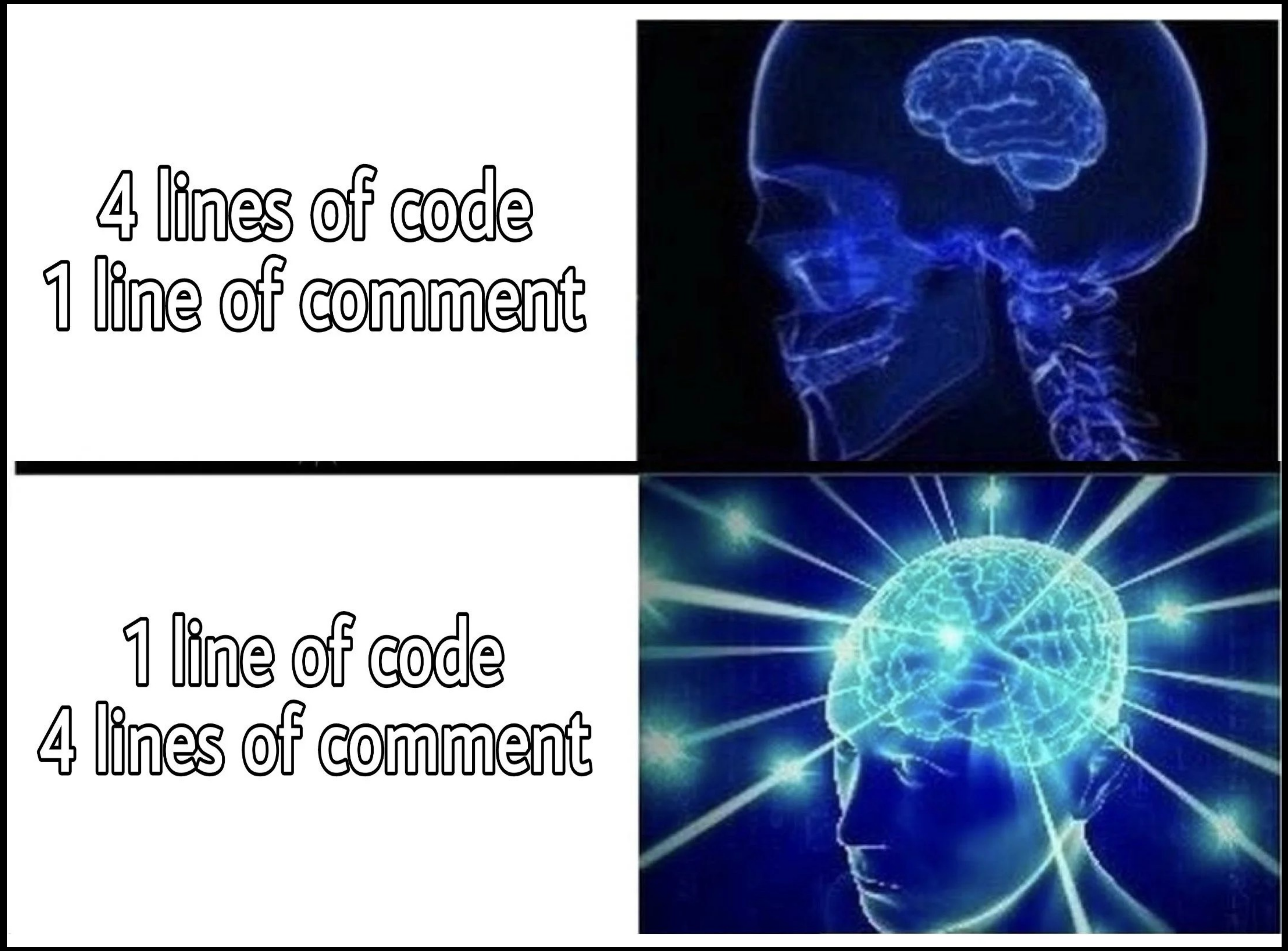
- Solutions? `Lazy<T>` for memoisation and `Maybe<T>` to distinguish `Some(null)` and `None`!

Motivation

- A better `InfiniteList<T>`:
-
  ```
  class InfiniteList<T> {
    private Lazy<Maybe<T>> head;
    private Lazy<InfiniteList<T>> tail;
    …
  }
  ```
- Delays computation until needed
- Allows you to infinitely generate data using:
  - Seed & transformer
  - Producer

Motivation

- `head` contain current value, `tail` contain future values
- Essentially a stream from CS1101S!
- Why do we need streams:
  - Pure functional programming paradigm languages e.g. Haskell
  - High-level implementation of concurrency/parallelism
  - And many more



Lab 7: `InfiniteList<T>`

- Basics:
  - Two factory methods to create an `InfiniteList`
    - `generate`: takes in a producer, generate using same producer
    - `iterate`: takes in a seed and a transformer, generate new values
  - `head` & `tail`: returns current value / generate next `InfiniteList`
  - `map`: modify values in the infinite list lazily
  - `filter`: filter out elements that fails the given predicate
    - Calling `tail` will eagerly evaluate until the next non-filtered value

Lab 7: `InfiniteList<T>`

- Static nested class `Sentinel`
  - Special tail to mark the end of a finite list
  - Factory method `sentinel()`
  - `toString()` returns “-“
  - `isSentinel()`: returns true if list is an instance of sentinel, false otherwise
  - You may cache a static final `SENTINEL`, just like `EMPTY` and `NONE`

Lab 7: `InfiniteList<T>`

- Not so basic:
  - `limit`
    - Limit the size of the stream and truncate
    - Filtered out elements should not count towards the limit
  - `toList`
    - Convert a stream to a list
    - Don’t need to care about infinite lists

Lab 7: `InfiniteList<T>`

- More difficult:
  - `takeWhile`
    - Ends the stream on the first value that the condition evaluates to false
    - Ignore filtered out elements
    - NOT terminal

Lab 7: `InfiniteList<T>`

- Terminal operations:
  - `count`
    - Self-explanatory, keep production until you reach the end
    - Returns a `long`
  - `reduce`
    - Works like `accumulate` in CS1101S
    - Takes in a combiner to combine all values in the infinite list

Lab 7: InfiniteList<T>

- Example:

    -
      ```
      InfiniteList.iterate(0, x -> x + 1)
                  .limit(5)
                  .reduce(0, (x, y) -> x + y)
      ```

    -
      ```
      InfiniteList.iterate(0, x -> x + 1)
                  .filter(x -> x % 2 == 1)
                  .limit(10)
                  .count()
      ```

    -
      ```
      InfiniteList.iterate(“A", s -> s + "Z")
                  .limit(2)
                  .map(s -> s.length())
                  .toList()
      ```

Grading Scheme

- 6% of overall grade
- Documentation: 2 marks
- Everything else: 22 marks
- Usual penalties apply
- Due 5 Apr (2-week deadline, but start early!)

About PE 2

- Overwhelmed `stu` at the beginning of PE 1
- `stu` went down for a long period before Lab 5 deadline
- Other way to access PE hosts besides tunnelling through `stu`:
  - SoC VPN (FortiClient VPN)
  - https://dochub.comp.nus.edu.sg/cf/guides/network/vpn
- SoC VPN ≠ NUS VPN
- If you're successfully connected, you should be able to directly connect by `ssh <username>@pe1xx.comp.nus.edu.sg`

Happy coding! 👨‍💻