NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

CS2030 — PROGRAMMING METHODOLOGY II
(Semester 2: AY2023/2024)
(ANSWERS)

Apr / May 2024 Time Allowed: 2 Hours

INSTRUCTIONS TO CANDIDATES

1. This assessment contains **TWENTY-TWO(22)** questions.
2. Answer **ALL** questions. The maximum mark is **40**.
3. This is an **EXAMPLIFY-SECURE OPEN-BOOK** assessment. You may refer to your lecture notes, recitation guides, lab codes, and the Java API.
4. You will be liable for disciplinary action which may result in expulsion from the University if you are found to have contravened any of the clauses below,
   - Violation of the NUS Code of Student Conduct (in particular the part on Academic, Professional and Personal Integrity), NUS IT Acceptable Use Policy or NUS Examination rules.
   - Possession of unauthorized materials/electronic devices.
   - Bringing your mobile phone or any storage/communication device with you to the washroom.
   - Unauthorized communication e.g. with another student.
   - Reproduction of any exam materials outside of the exam venue.
   - Photography or videography within the exam venue.
   - Plagiarism, giving or receiving unauthorised assistance in academic work, or other forms of academic dishonesty.
5. Once you have completed the assessment,
   - Click on the "Exam Controls" button and choose "Submit Exam".
   - Check off I am ready to exit my exam and click on "Exit" to upload your answers to the server.
   - You will see a green confirmation window on your screen when the upload is successful. Please keep this window on your screen.
   - If you do not see a green window, please disconnect and reconnect your WIFI and try again.
   - Please be reminded that it is your responsibility to ensure that you have uploaded your answers to the Software.

SECTION A (15 Multiple Choice Questions : 15 Marks)
One mark for each correct answer and no penalty for wrong answer. There are five options for each question. Choose "(E) None of the Above" only if the answer cannot be found in the first four choices.
You may assume that all code fragments are syntactically correct and compilable, and that any deviations detected are due to either minor typos or short-cuts and abbreviations used.

1. Consider the two class declarations below where method `mn` in class `B` *overrides* the corresponding method `mn` in class `A`.

```
class A {
    public t1 mn(t3 v) {
        ...
    }
}

class B extends A {
    public t2 mn(t4 v) {
        ...
    }
}
```

Which of the following typing conditions best satisfy Java's compile-time type checking for method overriding? Note that `t1<:t2` states that type `t1` is a subtype of type `t2`. Also, `t1=t2` is equivalent to `t1<:t2` and `t2<:t1`.

(i) `t1<:t2`
(ii) `t2<:t1`
(iii) `t3<:t4`
(iv) `t4<:t3`

(A) (i) and (iv)
(B) (ii) and (iii)
(C) (i), (iii) and (iv)
(D) (ii), (iii) and (iv) **(ANSWER)**
(E) None of the Above

Conditions for Method Overriding:

```
t3=t4 and t2<:t1
<=> t3<:t4 and t4<:t3 and t2<:t1
```

Hence, Answer is (D).

2. Consider the class declaration below where two methods of the same name `mn` in class `A` are expected to be *overloaded*.

```
class A {
    public t1 mn(t3 v) {
        ...
    }

    public t2 mn(t4 v) {
        ...
    }
}
```

Which of the following conditions best satisfy Java's compile-time type checking for method overloading?

(i) `not(t1<:t2)`

(ii) `not(t2<:t1)`

(iii) `not(t3<:t4)`

(iv) `not(t4<:t3)`

Choose one of the answers below.

(A) (i) or (ii)

(B) (i) and (ii)

(C) (iii) or (iv) **(ANSWER)**

(D) (iii) and (iv)

(E) None of the Above

Conditions for Method Overloading:

```
not(t3 = t4)
<=> not(t3<:t4 and t4<:t3)
<=> not(t3<:t4) or not(t4<:t3)
```

Hence, Answer is (C).

3. Consider the class declaration below

```
class P {
    private final int x;
    ...
}
```

Below are four statements about field `x`.

(i) field x can be modified inside any constructor of class `P`

(ii) field x can be modified inside any non-constructor method of class `P`

(iii) field x can be read inside any constructor of a sub-class of `P`

(iv) field x can be read inside any non-constructor method of a sub-class of `P`.

Which of the above statements are true?

(A) (i) **(ANSWER)**

(B) (i) and (ii)

(C) (i) and (iii)

(D) (i), (ii), (iii) and (iv)

(E) None of the Above

`private final` *field of class* `P` *can only be changed in the constructor of* `P` *when object is first created. Hence, Answer is (A).*

4. Consider the following Java program fragment:

```
class A {
    public void foo(A v) {
        System.out.print("1");
    }
}

class B extends A {
    public void foo(A v) {
        System.out.print("2");
    }

    public void foo(B v) {
        System.out.print("3");
    }
}

B b1 = new B();
B b2 = new B();
A a1 = b1;
A a2 = new A();
b1.foo(b2);
b1.foo(a1);
a1.foo(a1);
a1.foo(a2);
```

What output will be printed when the code fragment (after the two class declarations) is executed?

(A) 3211

(B) 3221

(C) 3222 **(ANSWER)**

(D) 3322

(E) None of the Above

Answer is (C) 3222.

```
b1.foo(b2);  // calls B.foo(B ..)
b1.foo(a1);  // calls B.foo(A ..)
a1.foo(a1);  // calls B.foo(A ..) based on runtime type of a1
a1.foo(a2);  // calls B.foo(A ..) based on runtime type of a1
```

5. Consider the following Java program fragment:

```
int i = 3;
double d = 3.2;
Integer I = 1;
```

Below are four statements about field `x`.

(i) `double e2 = I;`

(ii) `Double E1 = d;`

(iii) `Double E2 = i;`

(iv) `Double E3 = I;`

Which statements would cuase compile-time error(s) in Java?

(A) (i) and (iii)

(B) (ii) and (iv)

(C) (iii) and (iv) **(ANSWER)**

(D) (i), (iii) and (iv)

(E) None of the Above

```
pa3.java:82: error: incompatible types: int cannot be converted to Double
        Double E2 = i;

pa3.java:83: error: incompatible types: Integer cannot be converted to Double
        Double E3 = I;
```

Hence, Answer is (C).

6. Consider the following Java program fragment:

```
interface I<T> {
    boolean goo(T x);
}

static .. choose(.. a, .. b)  {
    if (a.goo(b)) {
        return a;
    }
    return b;
}
```

Which of the following is the *most general* Java type declaration for the `choose` method?

(A) `static <T> I<T> choose(I<T> a, I<T> b)`

(B) `static <T extends I<T>> T choose(T a, T b)`

(C) `static <T extends I<? extends T>> T choose(T a, T b)`

(D) `static <T extends I<? super T>> T choose(T a, T b)` **(ANSWER)**

(E) None of the Above

The most general type declaration for choose should be

```
static <T extends I<? super T>>  T choose(T a, T b)
```

We need `(T extends I<? super T>)` *due to* `a.goo(b)` *method call, where* `T` *of* `I<T>` *is being used as a function input. Hence, answer is (D).*

7. The `Iterator<E>` interface in Java has the following methods:

```
boolean hasNext() { .. }
E next() { .. }
void remove() { .. }
void forEachRemaining(Consumer<? super E>) action) { .. }
```

Which of these methods are expected to be pure functions and effects-free?

(A) `hasNext` **(ANSWER)**

(B) `next`

(C) `hasNext` and `next`

(D) `hasNext`, `next` and `forEachRemaining`

(E) None of the Above

Only `hasNext()` *is a pure function.*

- `next()` *may throw an exception;*
- `remove()` *changes the iterator;*
- `forEachRemaining`*'s* `action` *can have side-effects.*

Hence, answer is (A).

8. It is possible to define a static method `flatten` that can convert a nested list of type `ImList<ImList<T>>` into `ImList<T>`. Given a nested list `[[1,2],[],[3,4]]`, this method would convert it to `[1,2,3,4]`.

   What could be the missing code fragment of the `flatten` method below?

```
static <T> ImList<T> flatten(ImList<ImList<T>> xss) {
    return ...
```

   (A) `xss.map(xs -> xs)`
   (B) `xss.flatMap(xs -> xs)` **(ANSWER)**
   (C) `xss.flatMap(xs -> xs.map(x -> [x]))`
   (D) `xss.flatMap(xs -> xs.flatMap(x -> x))`
   (E) None of the Above

```
flatten(xss) = xss.flatMap(xs -> xs)
```

This is also a general method for any context `C<X>`

```
flatten : C<C<X>> -> C<X>
```

and is related to `flatMap` *as above. Hence, answer is (A).*

9. An instance method `g` can be implemented using another instance method `f`, if we could construct method `g` using `f`. That is, we can define `g`, as follows:

```
.. g(..) {
    return this.f(...)
}
```

Consider the API of the `ImList<T>` class below.

```
class ImList<T> {
    ImList<T> filter(Predicate<T> f) { .. }
    ImList<R> map(Function<? super T, ? extends R> f) { .. }
    ImList<R> flatMap(Function<? super T, ? extends ImList<R>> f) { .. }
    <R> ImList<R> reduce(R identity,
        BiFunction<? super R, ? super T, ? extends R> f) { .. }
}
```

Which of the following is said to be true about methods of the `ImList<T>` context.

(A) `filter` can be implemented using `flatMap` **(ANSWER)**

(B) `reduce` can be implemented using `map`

(C) `flatMap` can be implemented using `map`

(D) `reduce` can be implemented using `flatMap`

(E) None of the Above

`filter` *can be implemented in terms of* `flatMap` *as follows:*

```
xs.filter(pred) = xs.flatMap(x -> if pred.test(x) then [x] else [])
```

Hence, answer is (A).

10. Consider the following functions using the `Log` class that was implemented in Lab #5.

```
Log<Integer> addTwo(int x) {
    return Log.<Integer>of(x + 2, "addTwo");
}

Log<Integer> multThree(int x) {
    return Log.<Integer>of(x * 3, "multThree");
}

Log<Integer> hoo(Log<Integer> obj) {
    return obj.flatMap(x -> addTwo(x)).flatMap(r1 -> multThree(r1))
}
```

What output will be printed by JShell for:

```
hoo(Log.<Integer>of(2)).flatMap(r1 ->
    hoo(Log.<Integer>of(2)).flatMap (r2 ->
        Log.<Integer>of(r1 + r2)))
```

(A) `Log[12]: addTwo, multThree`

(B) `Log[24]: addTwo, multThree`

(C) `Log[24]: addTwo, multThree, addTwo, multThree` **(ANSWER)**

(D) `Log[78]: addTwo, multThree, addTwo, multThree`

(E) None of the Above

```
Log<Integer> hoo(Log<Integer> obj) {
    return obj.flatMap(x -> addTwo(x)).flatMap(r1 -> multThree(r1))
}
// this method would add "addTwo, multTree" to log
// and returns (x+2)*3
```

Thus,

```
hoo(Log.<Integer>of(2)).flatMap(r1 ->
    hoo(Log.<Integer>of(2)).flatMap (r2 ->
        Log.<Integer>of(r1 + r2)))
```

The first `hoo` *call would add* `"addTwo, multTree"` *and return* $r1 = (2 + 2) * 3 = 12$.

Similarly, the second `hoo` *call would add* `"addTwo, multTree"` *and return* $r2 = (2 + 2) * 3 = 12$.

Hence, final answer that is returned is (C).

`Log[24]: addTwo, multThree, addTwo, multThree`

11. Consider the following functions using the `Log` class that was implemented in Lab #5.

```
Log<Integer> addTwo(int x) {
    return Log.<Integer>of(x + 2, "addTwo");
}

Log<Integer> multThree(int x) {
    return Log.<Integer>of(x * 3, "multThree");
}

Log<Integer> hoo(Log<Integer> obj) {
    return obj.flatMap(x -> addTwo(x)).flatMap(r1 -> multThree(r1))
}
```

What output will be printed by JShell for:

```
Log<Integer> k = hoo(Log.<Integer>of(2))
k.flatMap(r1 -> k.flatMap (r2 -> Log.<Integer>of(r1 + r2)))
```

(A) `Log[12]: addTwo, multThree`

(B) `Log[24]: addTwo, multThree`

(C) `Log[24]: addTwo, multThree, addTwo, multThree` **(ANSWER)**

(D) `Log[78]: addTwo, multThree, addTwo, multThree`

(E) None of the Above

For this code we have

```
Log<Integer> k = hoo(Log.<Integer>of(2))
k.flatMap(r1-> k.flatMap (r2-> Log.<Integer>of(r1 + r2)))
```

Since this code is pure, it has the same answer as Q10 (C).

`Log[24]: addTwo, multThree, addTwo, multThree`

12. Consider the `Lazy` context with the following API.

```
class Lazy<T> {
    static <E> Lazy<E> of(E x) { .. }
    static <E> Lazy<E> of(Supplier<E> f) { .. }
    Lazy<Optional<T>> filter(Predicate<T> f) { .. }
    <R> Lazy<R> map(Function<? super T, ? extends R> f) { .. }
    <R> Lazy<R> flatMap(Function<? super T, ? extends Lazy<R>> f) { .. }
}

v1 = Lazy.of(() -> {System.out.print("eval 2;"); return 2;})
v2 = Lazy.of(() -> {System.out.print("eval 3;"); return 3;})
```

What output will be printed by JShell for:

```
lst = ImList.of(v1, v2);
lst.size() + lst.size();
```

(A) `4` **(ANSWER)**

(B) `4eval 2;eval 3;`

(C) `eval 2;eval 3;4`

(D) `eval 2;eval 3;eval 2;eval 3;4`

(E) None of the Above

For the code below

```
lst = ImList.of(v1, v2);
lst.size() + lst.size()
```

`v1` *and* `v2` *are* `Lazy` *objects (built via* `Supplier` *functions) that are placed inside an* `ImList`*. When the method* `size()` *of* `ImList` *is invoked, it does not need the values of its elements. Hence, the elements will not be evaluated.*

Hence, answer is (A).

13. Consider the `Lazy` context with the following API:

```
class Lazy<T> {
    static <E> Lazy<E> of(E x) { .. }
    static <E> Lazy<E> of(Supplier<E> f) { .. }
    Lazy<Optional<T>> filter(Predicate<T> f) { .. }
    <R> Lazy<R> map(Function<? super T, ? extends R> f) { .. }
    <R> Lazy<R> flatMap(Function<? super T, ? extends Lazy<R>> f) { .. }
}

v1 = Lazy.of(() -> {System.out.print("eval 2;"); return 2;})
v2 = Lazy.of(() -> {System.out.print("eval 3;"); return 3;})
```

What output will be printed by JShell for:

```
lst = ImList.of(v1, v2);
lst.map(x -> x.get()).size() + lst.map(x -> x.get()).size();
```

(A) `4`

(B) `4eval 2;eval 3;`

(C) `eval 2;eval 3;4` **(ANSWER)**

(D) `eval 2;eval 3;eval 2;eval 3;4`

(E) None of the Above

For the code:

```
lst = ImList.of(v1, v2);
lst.map(x-> x.get()).size() + lst.map(x-> x.get()).size()
```

`lst.map(x->x.get())` *would force the* `Lazy` *objects* `v1` *and* `v2` *to be evaluated. Hence, printing of* `"eval 2; eval 3"` *will first appear.*

The second `lst.map(x->x.get())` *would only retrieve the memoized (cached) values of* `v1` *and* `v2`*. Hence, there will be no printing of* `"eval 2; eval 3;"` *in the second call.*

14. Consider the following stream code fragment:

```
List<Double> lst = ..
Stream<Double> str = lst.stream()
str.reduce(1.0, (x,y) -> x / y)
```

The above reduction process is currently sequential. What would be a suitable parallelization for the above code fragment?

Note that in Answer (D) below, we use `(e1,e2)` as a short-hand to denote either a tuple of arguments or a `Pair` of values.

(A) `str.parallel().reduce(1.0, (x,y) -> x / y)`

(B) `str.parallel().map(x -> 1 / x).reduce(1.0, (x,y) -> x / y)`

(C) `str.parallel().map(x -> 1 / x).reduce(1.0, (x,y) -> x * y)` **(ANSWER)**

(D)
```
str.parallel().map(x ->(x, a -> 1 / a)).reduce((1.0, b -> b),
    ((a1,f1),(a2,f2)) -> (a1 * (f1.apply(a2)), f1.compose(f2))).first()
```

(E) None of the Above

This example is similar to Q1 of Recitation 9.

Since `x / y` *is not associative, it is not possible to directly parallelize the reduction of* `str.reduce(1.0, (x,y)-> x / y)`.

However, we can use the relationship `x / y = x * (y^(-1))`.

With this, left reduction of `(((1.0/x1)/x2)/x3)/x4` *is equivalent to*

`1.0*(x1^(-1)*(x2^(-1))*(x3^(-1))*(x4^(-1))`.

Hence, answer is (C).

```
str.parallel().map(x-> 1 / x).reduce(1.0, (x,y)-> x * y)
```

15. Given a simplified API for `CompletableFuture` (abbreviated as `CF`) shown below.

```
class CF<T> implements Future<T>, CompletionStage<T> {
    static <U> CF<U> supplyAsync(Supplier<U> s) { .. }
    <U> CF<U> thenApply(Function<T,U> f) { .. }
    <U> CF<U> thenComposeAsync(Function<T,? extends CF<U>> f) { .. }
    <U,V> CF<V> thenCombine<CF<U> u, BiFunction<T,U,V> f) {...}
    T join() { .. } // returns result when completed or an exception
}
```

Consider the following code fragment where variables `v1` to `v6` have been declared with an appropriate type.

```
v1 = CF.supplyAsync(() -> f1())
v2 = CF.supplyAsync(() -> f2())
v3 = CF.supplyAsync(() -> f3())
v4 = v1.thenApply(x -> { v2.join(); return f4(x); })
v5 = v3.thenComposeAsync(x -> f5(x))
v6 = v4.thenCombine(v5, (x,y) -> f6(x,y))
```

There are exactly six distinct calls `f1(..),..,f6(..)` in the above code fragment. Let us define `hb`(f_i, f_j) as a happens-before relation stating that method call $f_i(..)$ must complete before the start of $f_j(..)$ call. Which of the following is not a happens-before relationship for the above code execution?

(A) `hb(f1,f4)`

(B) `hb(f2,f4)`

(C) `hb(f2,f6)`

(D) `hb(f4,f6)`

(E) None of the Above **(ANSWER)**

For the `CF` *code below*

```
v1 = CF.supplyAsync(()-> f1())
v2 = CF.supplyAsync(()-> f2())
v3 = CF.supplyAsync(()-> f3())
v4 = v1.thenApply(x-> { v2.join(); return f4(x); })
v5 = v3.thenComposeAsync(x-> f5(x))
v6 = v4.thenCombine(v5, (x,y)-> f6(x,y))
```

We can state the dependency, as follows.

```
// v2.join();return f4(x)  means hb(f2,f4)
// v1.thenApply(..f4(x)..) means hb(f1,f4)
// v3.thenComposeAsync(x-> f5(x)) means hb(f3,f5)
// v4.thenCombine(v5, (x,y)-> f6(x,y)) mens hb(f4,f6) and hb(h5,f6)
// By transitivity of hb(f2,f4) and hb(f4,f6), we have hb(f2,f6).
```

Hence, answer in (E).

SECTION B (7 Questions: 25 Marks)
Unless otherwise specified, **do not** define your own classes or use classes other than those specified in the Java API (even if they have been defined in the course such as `ImList`, `Pair`, `Lazy`, `Try`, `Maybe`, `Log`, etc). You do not need to write `import` statements.

16. [**3 marks**] Study the following `Point` class.

```
class Point {
    private final double x;
    private final double y;

    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    double distanceTo(Point other) {
        return this.distanceTo(other.x, other.y);
    }

    double distanceTo(double a, double b) {
        double dx = this.x - a;
        double dy = this.y - b;
        return Math.sqrt(dx * dx + dy * dy);
    }
}
```

Objects can be modeled using closures of a local class. In particular, instance variables encapsulated in an object can be modeled as variable captures in a local class. Given the following `Point` interface,

```
interface Point {
    double distanceTo(double x, double y);
    double distanceTo(Point other);
}
```

complete the `point` method that takes in two `double` coordinates and returns an instance of a `Point`. A sample run is given below:

```
jshell> point(1.0, 1.0).distanceTo(point(2.0, 2.0))
$.. ==> 1.4142135623730951
```

ANSWER:

```
Point point(double x, double y) {
    return new Point() {
        public double distanceTo(Point p) {
            return p.distanceTo(x, y);
        }
        public double distanceTo(double px, double py) {
            double dx = x - px;
            double dy = y - py;
            return Math.sqrt(dx * dx + dy * dy);
        }
    };
}
```

The `x` *and* `y` *variables are captured in the local class definition of* `new Point() {...}`*. Since the captures are effectively final, they behave similarly to* `private final` *fields.*

The overloaded `distanceTo(Point p)` *method can only be defined by calling the* `distanceTo(double,double)` *method through* `Point p`*.*

Any reference to `p.x` *(and* `p.y`*) is not allowed as every* `new Point` *local class implementation is different, though they all implement the* `Point` *interface.*

17. [**3 marks**] This question is a continuation of question 16.

Write the `Circle` interface, as well as the `circle` method that takes in a point as the centre followed by a radius. Use the following sample run as a guide for your implementation.

```
jshell> Circle c = circle(point(0.0, 0.0), 1.0)
c ==> Circle with radius 1.0

jshell> c.contains(point(1.0, 1.0))
$.. ==> false

jshell> c.contains(point(0.5, 0.5))
$.. ==> true
```

ANSWER:

```
interface Circle {
    boolean contains(Point p);
}

Circle circle(Point centre, double radius) {
    return new Circle() {
        public boolean contains(Point p) {
            return centre.distanceTo(p) < radius;
        }

        public String toString() {
            return "Circle with radius " + radius;
        }
    };
}
```

Just like the previous question, the captured variables `centre` *and* `radius` *variables behave similarly to* `private final` *fields.*

The implementation of local class `new Circle` *should be straightforward. Don't forget the* `toString` *method too.*

18. [**3 marks**] Given the following `add` method:

```
int add(int a, int b) {
    return a + b;
}
```

To multiply $a \times b$ where $a > 0$ and $b > 0$, we can define a recursive `mul` method that makes use of the `add` method:

```
int mul(int a, int b) {
    if (b == 1) {
        return a;
    }
    return add(mul(a, b - 1), a);
}
```

Similarly, to compute the exponential a^b where $a > 0$ and $b > 0$, we can define a recursive `exp` method that makes use of the `mul` method:

```
int exp(int a, int b) {
    if (b == 1) {
        return a;
    }
    return mul(exp(a, b - 1), a);
}
```

By applying the *abstraction principle*, write a `hyper` method that defines a hyper-operation (such as `mul` and `exp`) that takes in two integer arguments `a` and `b`, as well as a `BinaryOperator<Integer>`, so as to abstract away the recursive computation.

Note that `BinaryOperator<T>` is syntactic sugar for a two-argument function `BiFunction<Integer,Integer,Integer>`.

ANSWER:

```
int hyper(int a, int b, BinaryOperator<Integer> f) {
    if (b == 1) {
        return a;
    }
    return f.apply(hyper(a, b - 1, f), a);
}
```

Note the similarity between the implementations of `mul` *and* `exp`*. Both* `mul` *and* `exp` *are recursive, so the only difference is that* `mul` *calls* `add`*, while* `exp` *calls* `mul`*. Following the abstraction principle, the common recursive structure can be "combined into one" (*`hyper`*), with the "varying part" abstracted out and passed to* `hyper` *via the* `BinaryOperator` *argument.*

19. [**2 marks**] This question is a continuation of question 18.

Given `add` as a two-argument function,

```
BinaryOperator<Integer> add = (a, b) -> a + b
```

define `mul` and `exp` as two-argument functions such that

```
jshell> add.apply(3, 3)
$.. ==> 6

jshell> mul.apply(3, 3)
$.. ==> 9

jshell> exp.apply(3, 3)
$.. ==> 27
```

Additionally, define `tet` as the (weak-)tetration operation given by $3 \downarrow\downarrow 3 = (3^3)^3$

```
jshell> tet.apply(3, 3)
$.. ==> 19683
```

ANSWER:

```
jshell> BinaryOperator<Integer> add = (x, y) -> x + y
add ==> $Lambda$..

jshell> BinaryOperator<Integer> mul = (x, y) -> hyper(x, y, add)
mul ==> $Lambda$..

jshell> BinaryOperator<Integer> exp = (x, y) -> hyper(x, y, mul)
exp ==> $Lambda$..

jshell> BinaryOperator<Integer> tet = (x, y) -> hyper(x, y, exp)
tet ==> $Lambda$..

jshell> add.apply(3,3)
$.. ==> 6

jshell> mul.apply(3,3)
$.. ==> 9

jshell> exp.apply(3,3)
$.. ==> 27

jshell> tet.apply(3,3)
$.. ==> 19683
```

This is how we can call `hyper` *by defining different lambda expressions for* `mul`, `exp` *and also* `tet`.

20. [**5 marks**] The following is a simplified `Log` class for logging computations.

```
1: class Log<T> {
2:     private final T t;
3:     private final String log;
4:
5:     private Log(T t, String log) {
6:         this.t = t;
7:         this.log = log;
8:     }
9:
10:     static <T> Log<T> of(T t, String log) {
11:         return new Log<T>(t, log);
12:     }
13:
14:     <R> Log<R> map(Function<? super T, ? extends R> mapper) {
15:         return new Log<R>(mapper.apply(this.t), this.log);
16:     }
17:
18:     <R> Log<R> flatMap(Function<? super T,
19:             ? extends Log<? extends R>> mapper) {
20:         Log<? extends R> logr = mapper.apply(t);
21:         return new Log<R>(logr.t, this.log + logr.log);
22:     }
23:
24:     String log() {
25:         return this.log;
26:     }
27:
28:     public String toString() {
29:         return this.t.toString();
30:     }
31: }
```

Notice that the output of the log only happens when the `log` method is invoked. However, logging is still performed eagerly.

Redefine the `Log` class such that logging is only performed when the `log` method is invoked. Within your answer, write ONLY the lines of code that are changed, together with the corresponding line numbers.

For example, if line `1` requires the generic type to be removed, then write

```
1: class Log {
```

ANSWER:

```
3:   private final Supplier<String> log;
5:   private Log(T t, Supplier<String> log) {
11:   return new Log<T>(t, () -> log);
21:   return new Log<R>(logr.t, () -> this.log.get() + logr.log.get());
25:   return this.log.get();
```

Make sure that constructors that build a new Logging string such as in Line 21 are changed. Line 21 is also needed to force the building/evaluation of a Logged string. Some answer that use `Lazy<String>` *instead of* `Supplier<String>` *are also acceptable.*

21. [**4 marks**] A pipe is a form of redirection that is used in Unix to send the output of one Unix command to the input of another. As an example, the following piped redirection

```
cat Program.java | grep return | wc -l
```

outputs the count of the number of lines (`wc -l`) that contains the word "return" (`grep return`) from the contents of the given file (`cat Program.java`). The input and output passed from one command to the next can be viewed as lines of text that make up a text file.

To simulate this process in Java, one can represent the pipeline as

```
cat("Program.java").grep("return").wc("-l")
```

where `cat` returns an instance of a `Unix` class with two methods `grep` and `wc` defined that also return `Unix` objects. Clearly, adding more commands will require that the `Unix` class be modified.

Alternatively, we can define `Unix` as a class that implements a `Piped` interface.

```
interface Piped {
    Piped pipe(Command command);
}
```

and write the pipeline as

```
cat("Program.java").pipe(grep("return")).pipe(wc("-l"))
```

In this way, new commands can be defined without modifying existing code.

Write your program in bottom-up dependency order. Leave out the `Piped` interface above and take note of the following:

- assume that `cat` has been written for you that returns a `Unix` object;
- each line of the text file is one element of an encapsulated `Stream<String>`;
- the pipeline should be lazily evaluated.

ANSWER:

```
interface Command extends UnaryOperator<Stream<String>>> { }

class Unix {
    Stream<String> text;

    Unix(Stream<String> text) {
        this.text = text;
    }

    Unix pipe(Command c) {
        return new Unix(c.apply(text));
    }

    public String toString() {
        return this.text.reduce("", (x,y) -> x + "\n" + y);
    }
}
```

Here, each object created in Unix will essentially contain an output file, denoted by a lazy stream of `String` *objects (lines), and you need a constructor for it.*

You can also have access to the lazy stream of strings via the `toString()` *method.*

Lastly, to compose commands for Unix, we must allow new Unix objects to be built via the pipe comand, e.g. `unx.pipe(grep("return"))` *would take a Unix object* `unx` *to pipe into the command* `grep("return")`*.*

Common Mistakes:

- *did not use* `Stream<String>` *text field*
- *did not define construtor*
- *used* `ImList<..>` *when* `Stream<String>` *is sufficient*
- *used* `Supplier` *when* `Stream` *is already lazy.* `Supplier` *is not needed*
- *unnecessary use of* `Optional`
- *missing* `Command extends UnaryOperator<Stream<String>>` *or* `Function<Stream<String>,Stream<String>>`
- *forgot to define* `toString`
- *use* `CompletableFuture<..>` *which is not necessary*

22. [**5 marks**] This question is a continuation of question 21.

Write the `grep` and `wc` methods according to the following sample run. You may find the `contains` method of the `String` class useful:

```
jshell> String s = "one two"
s ==> "one two"

jshell> s.contains("one")
$.. ==> true

jshell> s.contains("et")
$.. ==> false
```

Here is a sample run. Note that when `wc` is called with `"-l"`, it counts the number of lines, while `"-c"` counts the total number of characters. If the argument is invalid, the pipeline prior to the invalid operation is retained.

```
jshell> Unix cat() { // simplified cat method for testing
   ...>     return new Unix(Stream.of("one", "two", "one two"));
   ...> }
|  created method cat()

jshell> cat().pipe(grep("one"))
$.. ==>
one
one two

jshell> cat().pipe(grep("one")).pipe(wc("-l"))
$.. ==>
2

jshell> cat().pipe(grep("one")).pipe(wc("-c"))
$.. ==>
9

jshell> cat().pipe(grep("one")).pipe(wc("-c")).pipe(wc("-c"))
$.. ==>
1

jshell> cat().pipe(grep("one")).pipe(wc("-c")).pipe(wc("-"))
$.. ==>
9

jshell> cat().pipe(grep("three")).pipe(wc("-c"))
$.. ==>
0
```

ANSWER:

```
Command grep(String word) {
    return s -> s.filter(y -> y.contains(word));
}

Command wc(String argument) {
    if (argument.equals("-l")) {
        return s -> Stream.of(s.count() + "");
    }
    if (argument.equals("-c")) {
        return s -> Stream.of(
            s.map(x -> x.length()).reduce(0, (x,y) -> x + y) + "");
    }
    return s -> s;
}
```

Command Mistakes:

- *wrong/missing* `grep` *(must have* `filter`/`contains`*)*
- *wrong/missing* `wc("-l")`
- `wc("-l")` *must have* `s.count()` *(or length and* `reduce`*)*
- *partial* `wc("-c")` *missing* `x.length()`
- *using* `toString()`/`println`
- *using* `peek()`
- *generating (infinite list)*
- *mix up* `-l` *and* `-c`
- *did not use* `Stream.of` *for* `wc` *command*

— END OF PAPER —