Sem 1 2017/18:

Multiple Choice:

1 C correct-c

2 B wrong-w

3 B c

\*4 A w

5 B w

6 E c

7 D w -pure functions do not throw exceptions, change the environment in any way, and are a 1-1 mapping.

8 D c

\*9 X field is on the heap as a variable of a class, x is not stored at all. Func is local and is on the stack. As a reference, func refers to a object stored on the heap.

10 B w main may terminate without the thread doing printAsync(1) being done.

11 A c – Syntax. But will be run sequentially

12 B w- main thread does nothing.

13)

Pred<T> and (Predicate<T> pred1,Predicate<T> pred2){

return new Predicate<T>(x-> pred1.test(x) && pred2.test(x)); wrong syntax.

}

**Predicate <T> and (Predicate<T> pred1,Predicate<T> pred2){**

**Return x-> pred1.test(x)&&pred2.test(x);**

**}**

**OR**

**Predicate <T> and (Predicate<T> pred1,Predicate<T> pred2){**

**Return new Predicate<T>(){**

**Public Boolean test(T x){**

**Return pred1.test(x)&&pred2.test(x);**

**}**

**}**

**}**

14)

Class TextShortener implements SummeryStrategy{

@Override

String summarize(String text, int textLimit){

Return TextShortener.shorten(text,textLimit);

}

}

Call:

createSnippet(new TextShortener(random text, limit))

**WRONG**

**createSnippet(TextShortener::shorten)**

**OR**

**createSnippet((s,n)-> TextShortener.shorten(s,n));**

15)

i) 26

ii)

The output is different as the function used in the parallelism is not a associative ~~nor stateless~~ function.

As in parallelism, the computation of each element of the stream is done in a deque,

The computation distributes the elements to be processed to multiple threads which each start with the identity.

Due to the work being distributed in parallel, the identity is called in each thread. Hence resulting in a different output than that of the sequential processing.

16)

i/ii)

Class LazyInt{

Supplier<Integer> s;

LazyInt(Supplier<Integer> s){

This.s = s;

}

Int get(){  
 return s.get();

}

LazyInt map(Function<? Super Integer,Integer> mapper){

Return new LazyInt(()-> mapper.apply(s.get());

}

LazyInt flatmap(Function <? Super Integer, LazyInt> flatmapper){

Return new LazyInt(()-> flatmapper.apply(this.get()).get();

}

}

iii) Yes LazyInt is a functor as it is a wrapper class that facilitates mapping between different typings, limited to Integers.

It is also able to follow the 2 functor laws.

An identity mapping returns the same result:

LazyInt(()-> new Integer(1)).map( x->x)

Returns 1

And, is able to be composed

LazyInt(()-> new Integer(1)).map( x->x).map(y-> y+1)

Returns 2.

CORRECT

iv)

Yes LazyInt also acts as a monad as it has a flatmap function that works to unwrap classes from a functor mapping and rewrap the computation back into a single layer of wrapping

Left identity: Monad.of(x).flatMap(f) ≡ f.apply(x)

– Right identity: monad.flatMap(x -> Monad.of(x)) ≡ monad

– Associative: monad.flatMap(f).flatMap(g) ≡

monad.flatMap(x -> f.apply(x).flatMap(g))

+++++

NO as it does not have a unit operator

A Monad has:

1. A unit operator /a .of which defines a unit operation on the function.
2. Left identity Monad.of(x).flatMap(f) == f.apply(x)
3. Right Identity: Monad.flatMap(f).flatMap(g) == Monad
4. Associative:

monad.flatMap(f).flatMap(g) == Monad.flatmap(x-> x.apply(x).flatMap(g))

In words:

1. Has a .of meaning a mapping of any variable to a wrapped variable on the function.
2. Left identity meaning that applying any pure function to a wrapped value is equivalent to applying the function to the unwrapped value itself.
3. Right identity means to apply a identity function function would return the same monad itself

17)

Import java.util.\*

Class Element{

Public int row;

Public int col;

Public double value;

Element(int r,int c,double val){

Row = r;

Col = c;

Value = val;

}

Public double get(int row, int col){

Return value;

}

@Override

Equals(Obj o)

hashCode()

}

Class Matrix{

Class Pair{

Private final l;

Private final r;

Pair(int l,int r){

L = l;

R = r;

}

Equals(Object o){

if (o instance of Pair){

Pair p = (Pair) o;

Return (o.l == l && o.r ==r);

}

Else{

Return false;

}

}

HashMap< Pair,Element> hash

Matrix(){

Hash = new Hashmap<Pair,Element>();

}

Public void set(int row,int col,double val){

Hash.put(new Pair(row,col),new Element(row, col,val));

}

Public double get(int row, int col){

Optional<Element> e = Optional.ofNullable(hash.get(new Pair(row,col)));

Return e.orElse(0);

}

}

I choose to implement my key value mapping using a internal Pair class to ensure that I adhere to the solid principle of Single Responsibility. The matrix class should only have the responsibility to store and retrieve values from the hashmap. Similarly the element class should only perform tasks directly related to the field stored inside itself.

Hence I create a new Pair class to act as a value placeholder for the hashmap mapping.

Iv)

Yes an element should be defined as a inner class for a matrix as it should not exist outside of a Matrix as it is closely tied to the functionality of a matrix and does not exist outside of a Matrix. This follows the single responsibility principle.

No, this would violate the Dependency Inversion Principle as we would be unable to modifiy the inner workings of a Element without modifying the matrix outer wrapping class.

Also, such an inner class would also violate the O principle, as the Element class would not be open for extension as it is totally coupled with the Matrix Class.

Overall I think there are pros and cons with doing it either way, but defining the Element class as its own class may allow for better encapsulation and modularity for more flexible and ease of maintenance.