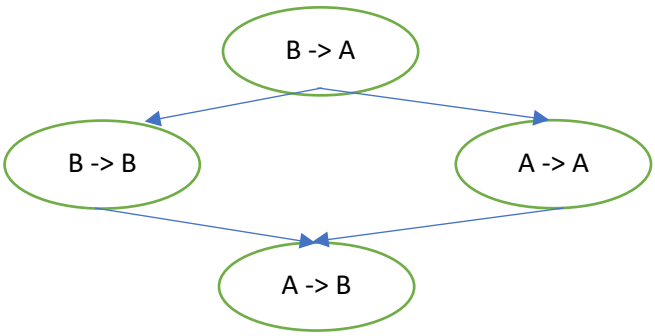
1. (a) Y f = (λh. (λx. h (x x)) (λx. h (x x))) f This represents infinite recursion of function f.
   (b) Y f = (λh. (λx. h (x x)) (λx. h (x x))) f
   = (λx. f (x x)) (λx. f (x x))
   = f ((λx. f (x x)) (λx. f (x x)))
   = f ((λx. h (x x)) (λx. h (x x)))
   = f (Y f)
   (c) Normal order evaluation: leftmost, outermost redex first.
   Applicative order evaluation: leftmost, innermost redex first.
   Where redex stands for reducible expression. An expression of the form (λx.E) M is a candidate for beta-reduction and is called a reducible expression.
   (d) (λx. + x x) ((λy. y) 3)
   Normal order:
   (λx. + x x) ((λy. y) 3) = + ((λy. y) 3) ((λy. y) 3)
   Applicative order:
   (λx. + x x) ((λy. y) 3) = (λx. + x x) 3
   (e) Church-Rosser theorem 1: Any two terminating reduction sequences starting with the same expression will result in the same normal form.
   Church-Rosser theorem 2: If any reduction sequence for an expression terminates, then normal order reduction will terminate.

2. (a) fun foo f g x = [((f (hd x)), (g (hd x)))]
   (b) ((‘a -> ’c) * (‘b -> ‘c)) -> ‘a -> ‘b list -> ‘c list
   (c) a and b are both functions, x is an element, (y::ys) is a list of elements, and the return type is also a list of elements. Assume x is of type ‘a, y is of type ‘b, and the return type is ‘c list. Then the input of a is of type ‘a and the output of a is of type ‘c, so a is of type (‘a -> ‘c), the input of b is of type ‘b and the output of b is of type ‘c, so b is of type (‘b -> ‘c), (y::ys) is of type ‘b list. Therefore. the type of the function is ((‘a -> ’c) * (‘b -> ‘c)) -> ‘a -> ‘b list -> ‘c list.

3. (a) The method to call is selected based on the actual type of the object, not the declared type of the variable.
   (b) A type is a set of values. A subtype defines a set of values that is a subset of the set defined by its parent type. Since B <: A, every object that has a type B also has a type A, but not vice versa.
   (c) Assuming B <: A, then we have covariance on the return type and contravariance on the argument type:



(d)

```
class A

class B extends A

def fa (g:A=>Int) = {
        val a:A = new A
        g(a)
}

def i(y:B) = 1

def main(args: Array[String]): Unit = {
        fa(i)
}
```

Explanation: B is a subtype of A. i is a function having B=>Int, fa is a function expecting as its argument a function A=>Int. If function subtyping is allowed covariantly, then B=>Int is a subtype of A=>Int. Therefore, we can pass i to fa and make the call fa(i). Then the function i (g in fa) will take an A as its parameter. Since i is expecting a B as input, B is a subtype of A, a B is also an A, but an A is not a B, the program will be unsafe.

4. (a) Assume A = Vehicle, B = Car, C = List, and B (Car) is a subtype of A (Vehicle).

```
public void AddVehicle(List<Vehicle> l){
        l.add(new Vehicle());
    }

List<Car> cl = new ArrayList<Car>();

AddVehicle(cl);

Car c = cl.get();
```

Explanation: List<Car> should not be a subtype of List<Vehicle>. If the compiler allows such covariant subtyping, then wherever the function is expecting a List<Vehicle> we can pass in a List<Car>. So we can pass a List<Car> into the function AddVehicle, which inserts a Vehicle into a list of Cars. If this were allowed, then if later we want to retrieve a Car from List<Car>, we get a Vehicle instead. Since we cannot assign a Vehicle to a Car variable, this will cause a runtime type error.

(b)

```
public <T> void AddElement(List<T> l, T t){
        l.add(t);
    }

    List<Car> cl = new ArrayList<Car>();
```

```
List<Vehicle> vl = new ArrayList<Vehicle>();

List<Object> ol = new ArrayList<Vehicle>();

Car c1 = new Car();

Vehicle v1 = new Vehicle();

Object o1 = new Object();

AddElement(cl);

AddElement(vl);

AddElement(ol);

Car c2 = cl.get();

Vehicle v2 = vl.get();

Object o2 = ol.get();
```

5. (a) Covariant subtyping: If C[A] is a generic class that parameterizes over a type parameter A, and S is a subtype of T, S <: T, then C[S] is a subtype of C[T], C[S] <: C[T].
   (b) Contravariant subtyping: If C[A] is a generic class that parameterizes over a type parameter A, and S is a subtype of T, S <: T, then C[T] is a subtype of C[S], C[T] <: C[S].
   (c)

```
[i]
abstract class myList[T]
case class List[T] (x:T, xs:myList[T]) extends myList[T]
case class Empty[T] () extends myList[T]
[ii]
def map [T, S] (f:T=>S, l:myList[T]) : myList[S] = {
        l match {
                case List(x, xs) => List(f(x), map(f, xs))
                case Empty() => Empty()
        }
}
```

6. (a) For reference counting collector, the cost of garbage collection is spread over the whole execution, so the program tends not to be interrupted for long. While for a mark and sweep collector, program suspends during garbage collection.
   (b) A copying garbage collector can use a heap pointer for heap allocation, while a mark/sweep collector requires the free list for heap allocation. The heap pointer is faster for heap allocation than the free list. Also, the cost of a copying garbage collector is O(L), which is proportional to the amount of live storage, while the cost of a mark/sweep garbage collector is O(M), which is proportional to the size of the heap.
   (c) Generational Copying Garbage Collection is like Copying garbage collection, but instead of two heaps, it used multiple heaps based on the survival time of objects. Each heap contains objects of

similar age, and these heaps are called generations. When a heap fills up, the live object in that heap is copied to the next (older) heap.

(d)
```
def delete (x):
        x.refcount = x.refcount – 1
        if x.refcount = 0:
            for p in x.pointers:
                delete(p)
            addToFreeList(x)
```