1.

(a) *As we discussed in class, the expression (λx. (x x)) (λx. (x x)) has no normal form. Write another expression that has no normal form. Make sure that your expression is distinct from (λx. (x x)) (λx. (x x)), i.e. that it wouldn't be convertible to (λx. (x x)) (λx. (x x)). Hint: Think about how you'd write a non-terminating expression in a functional language.*

(λx. (x x x)) (λx. (x x x))

This expression never terminates. Trying to apply beta convertion makes it (λx. (x x x)) (λx. (x x x)) (λx. (x x x)). Applying beta conversion again makes it grow even more.

(b) *Write the definition of a recursive function (other than factorial) using the Y combinator. Show a series of reductions of an expression involving that function which illustrates how it is, in fact, recursive (as I did in class for factorial).*

Recursive function for computing sum to n terms using Y combinator will be

Y(λf. λn. if (= n 1) 1 (+ n (f (- n 1))))

Let's call the above function as SUM

SUM = Y(λf. λn. if (= n 1) 1 (+ n (f (- n 1))))

Computing for n = 3, the computing happens as follows

Y(λf. λn. if (= n 1) 1 (+ n (f (- n 1)))) 3

From the property of combinatory function,

Yf = f(Yf)

(λf. λn. if (= n 1) 1 (+ n (f (- n 1))))(Y(λf. λn. if (= n 1) 1 (+ n (f (- n 1)))))3

Applying beta conversion,

λn. if (= n 1) 1 (+ n (SUM(- n 1)))3

Applying beta conversion

If(= 3 1) 1 (+ 3 (SUM(-3 1)))

Applying gama conversion

3 + SUM(2)

3 + Y(λf. λn. if (= n 1) 1 (+ n (f (- n 1)))) 2

This continues until n is equal to 1. The final expression reduces to

3 + 2 +1 = 6

(c) *Write the actual expression in the λ-calculus representing the Y combinator, and show that it satisfies the property Y(f) = f(Y(f)).*

The Y combinator function Y is given by

(λh. (λx. h(x x)) (λx. h(x x)))

The property is

Yf = f(Yf)

Now, Yf can be written as

Yf = (λh. (λx. h(x x)) (λx. h(x x)))f

Applying beta conversion

Yf = (λx. f(x x)) (λx. f(x x)) ----------(1)

Applying beta conversion

Yf = f((λx. f(x x)) (λx. f(x x)))

From 1, we have

Yf = f(Yf)

(d) *Summarize, in your own words, what the two Church-Rosser theorems state*

The two Church Rosser Theorems are:
Theorem 1: Given any expression, no matter what the order of reduction is, the final result will be the same normal form of that expression. The ordering in which the reduction are choose does not make a difference to the eventual results.
Theorem 2: Given any expression, if any reduction order terminates then normal order reduction will also terminate. It is safe to assume that if an expression I can be reduced to normal form then it can be reduced so using normal order reduction.

2.

(a) *In ML, why do all lists have to be homogeneous (i.e. all elements of a list must be of the same type)?*

ML performance type checking. A homogenous list aids in type checking. If lists were to be the non-homogenous then any type checking on anything that involves lists cannot be done.

(b) *Write a function in ML whose type is (’a -> ’b) -> (’b list -> ’c list) -> ’a -> ’c.*

```
fun someFunction f g a =
        hd (g [(f a)])
```

This function takes 3 parameters:
Two functions:

i. f : a -> b
ii. g: b list -> c list

Third parameter

a: a

The return type is c

(c) *What is the type of the following function (try to answer without running the ML system)?*

```
fun foo f (op >) x (y,z) =
let
        fun bar a = if x > z then y else a
in
        bar [1,2,3]
end
```

fn : 'a -> ('b * 'c -> bool) -> 'b -> int list * 'c -> int list

(d) *Provide an intuitive explanation of how the ML type inference would infer the type that you gave as the answer to the previous ques*

Let f be of type a
Let (op >) be a function that take tuple b * c and returns bool(since it is being used in if condition)
x will be of type b since it is being compared(op >).
The other parameter for (op >) is z so, z is of type c
And bar function returns y or a. a is of type int list which means y should also be int list
(y,z) be of type int list * c
The return type of the entire function is int list since it is the return type of bar function.

Therefore, it is

fn: a->(b*c -> bool) -> b -> int list * c -> int list

3.

Consider the following package specification for an Ada package that implements a queue of integers.

```
package queue is
        function extract return integer;
        function insert(x: integer);
end queue;
```

(a) *Why would this package not be said to implement an abstract data type (ADT) for a queue?*

A package is not an abstract data type because in ada for a package to be called ADT, there needs to be a representation of data(type). And its data type should be used in all sub programs.

(b) *Modify the above package specification, and implement a simple package body (that performs no error checking), so that a queue is an ADT.*

Package Specification:

```
package adt_queue is
        type queue is private;
        procedure extract(q: in out queue; x: out integer);
        procedure insert(q: in out queue; x: in integer);
        procedure initialize(q: in out queue);
        private
              type int_array is array(1..20) of integer;
              type queue is record
                 the_queue: int_array;
                 startIndex: integer;
                 endIndex: integer;
             end record;
end adt_queue;
```

Package Body:

```
package body adt_queue is
        procedure extract(q:in out queue; x:out integer) is
        begin
            x :=  q.the_queue(q.startIndex);
            q.startIndex = q.startIndex +1;
            if (q.startIndex > 20) then
                q.startIndex = 1;
        end;

        procedure insert(q: in out queue; x: in integer) is
        begin
            q.the_queue(q.endIndex) := x;
            q.endIndex := q.endIndex + 1;
            if (q.endIndex > 20) then
                q.endIndex = 1;
        end;

        procedure initialize(q:in out queue) is
        begin
            q.startIndex := 0;
            q.endIndex := 1;
        end;
end adt_queue;
```

4.

(a) *As discussed in class, what are the three features that a language must have in order to be considered as object oriented?*

The three features are:

(i) Encapsulation of data and code
Eg: Fields and methods of a class are encapsulated in a class

(ii) Inheritance
Eg: The new class has methods and data of base class along with its own data and methods.

(iii) Subtyping with dynamic dispatch
Subtyping: One type(sub type) is considered to be of another type(super type).
Dynamic dispatch: Methods are invoked according to the actual type of the objects.
Eg: void foo(A val)
{

```
        val.toString();
    }
```

Assume B derives from A. We can either pass B or A object to foo. The toString that is called depends on the object that is passed.

(b)

i. *What is the "subset interpretation of suptyping"?*

A child class defines a set of objects which is subset of set defined by parent class.

ii. *Provide an intuitive answer, and give an example, showing why class derivation in Java satisfies the subset interpretation of subtyping.*

```
Class Vehicle
{
        int posX,posY;
        void move(int x, int y)
        {---}
}

Class car extends Vehicle
{
        Int numWheels;
        void move(int x, int y)
        {---}
}

void foo( Vehicle V)
{
        V.move(100,100);
}

Vehicle vObject = new Vehicle();
Car cObject = new Vehicle();

foo(vObject);    -- will work as foo is expecting an object of vehicle class
foo(cObject);    -- will work as Car <: Vehicle and Car object is a type of Vehicle object only
```

Thus, Java satisfies the subset interpretation of subtyping.

iii. *Provide an intuitive answer, and give an example, showing why subtyping of functions in Scala satisfies the subset interpretation of subtyping.*

```
class A(x:Int){
        val value = x;
```

```
	override def toString()= "A["+value+"]";
}

class B(x: Int) extends A(x){
	override def toString()= "B["+value+"]";
}

def foo(a: A){
	println(a.toString());
}

val AObject = new A(5)
val BObject = new B(10);
```

foo(AObject) -- Will work and print "A[5]" as foo is expecting an A class object
foo(BObject) -- Will work and print "B[10]" as foo is expecting an A class object and an B object is a type of A object

(c) *Consider the following Scala definition of a tree type, where each node contains a value.*
abstract class Tree[T <: Ordered[T]]
case class Node[T <: Ordered[T]](v:T, l:Tree, r:Tree) extends Tree[T]
case class Leaf[T <: Ordered[T]](v:T) extends Tree[T]
Ordered is a built-in trait in Scala (see http://www.scala-lang.org/api/current/index.html#scala.math.Ordered). Write a Scala function that takes a Tree[T], for any ordered T, and returns the maximum value in the tree. Be sure to use good Scala programming style.

```
def maxValue[T <: Ordered[T]] (myTree:Tree[T]) : T = myTree match{
	case Leaf(x) => x
	case Node(x, left, right) =>
		if (x > maxValue(left) && x > maxValue(right))
			x
		else {
			if(maxValue(left) > maxValue(right))
				maxValue(left)
			else
				maxValue(right)
			}
}
```

(d) *In Java generics, subtyping on instances of generic classes is invariant. That is, two different instances C<A> and C<B> of a generic class C have no subtyping relationship, regardless of a subtyping relationship between A and B (unless, of course, A and B are the same class).*

i. *Write a function (method) in Java that illustrates why, even if B is a subtype of A, C<B> should not be a subtype of C<A>. That is, write some Java code that, if the compiler allowed such covariant subtyping among instances of a generic class, would result in a run-time type error.*

```
class A {

}

class B extends A{

}

public class Assignment {

	public static void insert(LinkedList<A> ListofA,A element){
		ListofA.add(element);
	}
	public static void main(String[] args) {

		LinkedList<A> ListA = new LinkedList<A>();
		LinkedList<B> ListB = new LinkedList<B>();
		A a = new A(2);

		insert(ListB,a); //Gives compile time error

		B b = ListB.get(0); //No compile time error

	}

}
```

In the above code, function insert adds an A class object to the LinkedList that contains objects of type A, but if java allowed covariant sub typing then insert(ListB,a) then it would have inserted “a” A type object in List of B objects. And then if we extract element of ListB into an object “b” of B type, it would have resulted in Run time error beacuse we cannot assign “a” to “b” as B<:A thus Java doesn’t allow such covariant subtyping among instances of a generic class.

ii. *Modify the code you wrote for the above question that illustrates how Java allows a form of polymorphism among instances of generic classes, without allowing subtyping. That is, make the function you wrote above be able to be called with many different instances of a generic class.*

```
public class Assignment {

	public static <T> void insert(LinkedList<T> ListofT,T element){
		ListofT.add(element);
	}
	public static void main(String[] args) {

		LinkedList<A> ListA = new LinkedList<A>();
		LinkedList<B> ListB = new LinkedList<B>();
```

```
            A a = new A(2);
            B b = new B(2,2);

            insert(ListA,a);
            insert(ListB,b);
            insert(ListB,a);   //Compile time error

            A a1 = ListA.get(0);
            B b1 = ListB.get(0);

        }

}
```

Thus the above code allows calling insert function on different types of Lists without allowing subtyping.

(e)

i. *In Scala, write a generic class definition that supports covariant subtyping among instances of the class. For example, define a generic class C[E] such that if class B is a subtype of class A, then C[B] is a subtype of C[A].*

```
class C[+T](x:T){
override def toString() = "C of "+x.toString()
}
```

ii. *Give an example of the use of your generic class.*

```
class A{
override def toString() = "A";
}
class B extends A{
override def toString() = "B";
}
def foo(x: C[A]) {
println(x.toString());
}
Val objA = new C[A](new A());
Val objB = new C[B](new B());
```

foo(objA) //prints “C of A”

foo(objB) //prints “C of B”

Even if the function foo is expecting a C[A] object, passing C[B] object will work in the above case as we have used covariant subtyping. B<:A => C[B]<:C[A] in this case

(f)

i. *In Scala, write a generic class definition that supports contravariant subtyping among instances of the class. For example, define a generic class C[E] such that if class B is a subtype of class A, then C[A] is a subtype of C[B].*

```
class C[-T](x:T){

override def toString() = "C of "+x.toString()

}
```

ii. *Give an example of the use of your generic class.*

```
class A{

override def toString() = "A";

}

class B extends A{

override def toString() = "B";

}

def foo(x: C[B]) {

println(x.toString());

}

Val objA = new C[A](new A());

Val objB = new C[B](new B());
```

foo(objB) //prints “C of B”

foo(objA) //prints “C of A”

Even if the function foo is expecting a D[B] object, passing D[A] object will work in the above case as we have used contravariant subtyping. B<:A => D[A]<:D[B] in this case.

5.

(a) *What is the advantage of a mark-and-sweep garbage collector over a reference counting collector?*

The advantages of mark and sweep over reference counting collector is the ability to work on cycles in the graph that is generated by heap objects. All references are edges. Reference counting collector fails to clean up nodes that are part of a cycle.
Also, there are chances of overflow of the reference count variable in reference counting collector.

(b) *What is the advantage of a copying garbage collector over a mark and sweep garbage collector?*

The major advantage of Copying GC is that it's faster than Mark & Sweep GC.
O(Mark & Sweep GC) = O(live objects' size) + O(heap size)
O(Copying GC) = O(live object's size)
Mark and sweep requires to use free list. This makes it expensive to find free block. However, with copying garbage collector that makes use of heap pointer method the allocation is quicker.

(c) *Write a brief description of generational copying garbage collection.*

Generational Copying GC is based on the idea that the older an object gets, the longer it can be expected to live.
In generational copying garbage collection, there are more than 2 heaps. New objects are allocated in the zero generation heap. When a generation x is full and GC needs to be run, then all the live nodes in generation x are moved to generation x+1. When the last generation is full and cannot find a free space in it then it means we are limited on heap space.

(d) *Write, in the language of your choice, the procedure delete(x) in a reference counting GC system, where x is a pointer to a structure (e.g. object, struct, etc.) and delete(x) reclaims the structure that x points to. Assume that there is a free list of available blocks and addToFreeList(x) puts the structure that x points to onto the free list.*

```
void delete(struct obj *x)
{
        x->refcount = x->refcount – 1;

        if ( x->refcount == 0)
        {
                For each x->child
                {
                        delete(x->child);
                }

                addToFreeList(x);
        }
}
```