1. (a) (define (fib n)
(cond ((= n 0) 0)
((= n 1) 1)
(else (+ (fib (- n 1)) (fib (- n 2)))))))

(b) (define (lfib n)
(letrec
((inner (lambda (n x y)
(cond
((= n 0) a)
(else (inner (- n 1) y (+ x y)))))))
(inner n 0 1)))

2. (a) $(\lambda_x . 7)((\lambda_x . x\ x)\ (\lambda_x . x\ x))$

(b) function Sum(n) = 1 + 2 +3 + …+n

Sum = $Y(\lambda_f. \lambda_n.$ if (= n 1) 1 (+n f(-n 1)))

Sum 5 = $Y(\lambda_f. \lambda_n.$ if (= n 1) 1 (+n f(-n 1))) 5

= $\lambda_f. \lambda_n.$ if (= n 1) 1 (+n f(-n 1)) $Y(\lambda_f. \lambda_n.$ if (= n 1) 1 (+n f(-n 1))) 5

= $\lambda_n.$ if (= n 1) 1 (+n (Y($\lambda_f. \lambda_n.$ if (= n 1) 1 (+n f(-n 1))) (- n 1)) 5

= if (= 5 1) 1 (+ 5 (Y($\lambda_f. \lambda_n.$ if (= n 1) 1 (+n f(-n 1))) (- n 1)))

= (+ 5 (Y($\lambda_f. \lambda_n.$ if (= n 1) 1 (+n f(-n 1)))) 4

= …=

(c) $Y = \lambda_h . (\lambda_x . h(x\ x))(\lambda_x . h(x\ x))$

$Y(f) = (\lambda_h . (\lambda_x . h(x\ x))\ (\lambda_x . h(x\ x)))\ f$

$\xrightarrow{\beta}$ $((\lambda_x . f(x\ x))\ (\lambda_x . f(x\ x)))$

$\xrightarrow{\beta}$ $f\ ((\lambda_x . f(x\ x))\ (\lambda_x . f(x\ x)))$

$= f(Y(f))$

(d) The Church-Rosser theorem 1 states that the order of the reductions is chosen doesn't make different result, when applying λ-calculus to expression. One expression has one normal form at most, even reduced by different order. The expression will get the same normal form either reduced by applicative order or normal order.

The Church-Rosser theorem 2 states that if one expression can be reduced to be normal form, then the normal reduction can reduces the expression to the normal form. However, applicative order evaluation may not terminate.

3.(a) ML is static typing. If a program passes a static type-checker, the type will be checked at compile-time. The all-possible input of the list must satisfy type-safety property (the same type).

(b) fun foo f g x = hd (g (hd (f x)));

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

(d) function foo is a curried function, there are three arguments and one return value: fn : bool -> ‘a -> ‘c -> ‘d.

Since the type of x, y are not constraints, so the type of x, y are x: ‘a and y: ‘b. According to the if statement (comparing x with y), we can get the type of operator > is (‘a * ‘b -> bool). And the third argument is a tuple (y, z). While from the return statement, we can know the return value is either z or a. The type of a is int list, so it infers that the type of return value and z is int list. So the third argument (y, z) is (‘b * int list). In conclusion, the type of the foo function is val foo = fn : ('a * 'b -> bool) -> 'a -> 'b * int list -> int list.

4. (a) The three features of objected oriented programming are:

1) Encapsulation of data and code: classes;

2) Inheritance: This allows classes to be arranged in a hierarchy that represents “is a type of”. Creating a new type (class) using code from an existing type.

3) Subtyping/ dynamic dispatch

Subtyping: A type can be used as if it were another type. (If type B is a subtype of type A, then any value of type B can be used as a value of type A.)

Dynamic dispatch: resolving the code to execute on an object based not on the declared type of a variable, but rather the actual object type.

(b)

I. If B is subtype of A, then B is a subset A.

II. Contravariant in the parameter type:

Example:

```
Var q: car -> int
Function r (y: BMW): int {…}
Function s (y: Vehicle): int {…}
```

BMW <: car , car <: Vehicle

q := r is not safe. Since q needs a car as parameter, r needs a BMW as parameter. BMW is a car, but a car may not be BMW, it might be other brands of cars. So passing subclass is not safe. While q:= s is safe, it because that s expects a vehicle, car is vehicle, so passing a car to the function needs vehicle is safe. In conclusion, function subtyping must be contravariant in the parameter type.

Covariant in the result type:

Example:

```
Class MountainBicycle extends Bicycle {
    MountainBicycle  getBicycle() {
        return new MountainBicycle();
    }
}
```

A subclass can override the getBicycle method to return a more specific type. Since MountainBicycle is bicycle, so method can return a mountain bicycle when returning bicycle. That is covariant in the result type. But we can’t return a vehicle (it might be a car or airplane) when a bicycle is requested to return.

III. According to the definition of the subset of subtyping, if B is a subtype of A, then the set of functions who take A as input is the subset of the set of functions who take B as input. And the set of functions whose return type is B is a subset of the set of functions whose return type is A. It means that the set of functions who take type A as input and output is type B will be the subset of the set of functions who take type B as input and output is type A. That is A => B ⊆ B => A. So function subtyping satisfies the subset interpretation of subtyping.

VI. Contravariant on parameter type:

```
class Animal
class Cat extends Animal
object contravariantExample {
        def  f (g: Cat => String) {
            val c: Cat = new Cat()
            g(c)
            }
        def h (x: Animal) = x.toString()
        def mian (args: Array [String]) {
            f(h)
            }
}
```

Covariant on the result type:

```
class Animal
class Tiger extends Animal
class Zoo[+A]
def foo(x: Zoo[Animal}) : Zoo[Animal] = identity(x)
foo(new Zoo[Tiger])
```

(c)

```
def minimumValue (t: Tree) : Tree = t match {
    case Node (v, l, r) => {
        if (minimumValue(l) < v){
            if (minimumValue(l) < minimumValue(r)) {
                return minimumValue(l)
            } else { return minimumValue(r)}
    } else {
            if (v < minimumValue(r)){
                return v
            } else {return minimumValue(r)}
        }
    }
   case Leaf (v) => v
}
```

(d)I. class Fruit {}
class Apple extends Fruit{}
class C <T> {}
public class Generic {
public static void main () {
C<Apple> apple = new C<Apple> ();
C<Fruit> fruit= apple;
}
}

II. class Fruit {}
class Apple extends Fruit{}
class C <T> {}

public class Generic {
public static void main () {
C<? extends Apple> apple = new C<Apple> ();
C<? extends Fruit> fruit = new C<Fruit> ();
fruit = apple;
}
}

(e)

1) covariant : class C[+A]
example:
class Fruit
class Apple extends Fruit
class C[+A](x: A) {
def f () = x
}

2) contravariant: class D[-B]
example:
class D[-B] {
def f(x: B) = 1
}

5.

(a) The advantage of Reference Counting:
Storage reclamation is incremental
- Happens a little bit at a time.
- The program will not be interrupted for long.
  - Suitable for real-time programs

The disadvantage of Mark/Sweep Collection:
- Program suspends during GC.
  - Unsuitable for real-time programs

(b) Unlike mark/sweep GC, the cost of Copying Collection is proportional to the amount of live storage, not the size of the heap. Cost of Mark/Sweep is proportional to the size of the heap, no matter how many of live objects there are.
(c) Generational Copying GC:
Generational Copying GC has number of heaps. The heaps are called generations: young generations, middle-aged generations, and old generations. The important objects that live longer stay in old generations. More objects will be garbage in the younger generations. Objects in one generation that are traversed during garbage collection are copied into the next older generation.
(d)
```
class Pointer {
    Int referenceCount;
    List<Pointer> referenceField
}

public static void delete (Pointer p)
{
    p.referenceCount = p.referenceCount – 1;
    if (p.referenceCount == 0)
        {for (int i = 0; i < p. referenceField.size(); i++)
            delete(p. referenceField.size(i))
            addToFreeList(p)
        }
}
```