Type Class

```
data Stack a = Stack [a]
push :: Stack a -> a -> Stack a
push (Stack xs) x = Stack (x:xs)
pop :: Stack a -> Stack a
pop (Stack []) = Stack []
pop (Stack xs) = Stack (let (_:xs') = xs in xs')
-- OR: pop (Stack xs) = Stack (tail xs)
```

We define a class named Collection and push, pop are its allowed operations.

```
class Collection c where
  push :: c -> Int -> c
  pop :: c -> c
```

We define a data type stack, with constructor Stack that takes a list of Int

```
data Stack = Stack [Int] deriving Show
```

We declare for type Stack to use Collection's operations and implement push and pop for Stack(kinda like implement an interface.

```
instance Collection Stack where
  push (Stack xs) x = Stack (x:xs)
  pop (Stack s) = case s of
                    [] -> Stack []
                    (x:xs) -> Stack xs
```

We create the queue data type

```
data Queue = Queue [Int] deriving Show
```

We implement Collection interface for Queue

```
instance Collection Queue where
  push (Queue xs) x = Queue (xs ++ [x])
  pop (Queue []) = Queue []
  pop (Queue q) = case q of
                    [] -> Queue []
                    (x:xs) -> Queue xs
```

Implement interface on more complicated data structures

Eq is the interface, == is its operation

```
class Eq a where
  (==) :: a -> a -> Bool
```

Int implements Eq

```
instance Eq Int where
  (==) = intEq     -- intEq primitive equality
```

In order for tuple(a,b) to implement Eq, we also need to constrain that a, b to implement Eq themselves

```
instance (Eq a, Eq b) => Eq (a,b) where
  (u,v) == (x,y)     = (u == x) && (v == y)
```

In order for list [a] implements Eq, a itself has the constraint that it has to implement Eq

```
instance Eq a => Eq [a] where
  (==) []     []     = True
  (==) (x:xs) (y:ys) = x==y && xs == ys
  (==) _      _      = False
```

Dictionary Passing Style

Suppose we are interested in considering two Haskell Ints i and j equal if the absolute value of i is equal to the absolute value of j:

```
abs i == abs j
class MyEq a where
   (===) :: a -> a -> Bool
```

Using an instance declaration, we can make Int an instance of this type class:

```
  instance MyEq Int where
    i === j = abs i == abs j
```

When processing the type class declaration for MyEq, the Haskell compiler will generate the following internal type and function declarations:

```
data MyEqD a = MkMyEqD (a -> a -> Bool)
(===) :: MyEqD a -> a -> a -> Bool
(===) (MkMyEqD eq) = eq
```

The data type `MyEqD` represents the dictionary for the type class `MyEq`, parameterized by a. Each such dictionary stores the implementation for the === operation for a particular type. `MkMyEqD` is a constructor that takes a function. The function === takes a dictionary and extracts the corresponding implementation from the dictionary/returns the functions.

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
deriving Show
```

make Trees an instance of the type class MyEq:

```
instance MyEq a => MyEq (Tree a) where
  (===) (Leaf v1) (Leaf v2) = v1 === v2
  (===) (Node v1 tl1 tr1) (Node v2 tl2 tr2) = v1
=== v2 && tl1 === tl2 && tr1 === tr2
  (===) _ _ = False
```

From such an instance declaration, the compiler will generate code to construct Tree dictionaries for MyEq.

```
dMyEqTree :: MyEqD a -> MyEqD (Tree a)
dMyEqTree elDict = MkMyEqD myEqTree
  where
       myEqTree (Leaf v1) (Leaf v2) = (===) elDict
v1 v2
       myEqTree (Node v1 tl1 tr1) (Node v2 tl2
tr2) = (===) elDict v1 v2 &&
       myEqTree tl1 tl2 &&
       myEqTree tr1 tr2
       myEqTree _ _ = False
```

`dMyEqTree` takes a `MyEqD` a dictionary and constructs a new `MyEqD (Tree a)` dictionary, using `MkMyEqD` constructors and the `myEqTree` function. Remember a dictionary is just storing functions.

`myEqTree` is a function that checks if two trees are equal.

`(===) elDict` extract `eq` from dictionary elDict

The `cmp` function below compares two values from any type that belongs to the `MyEq` type class and returns a String indicating whether the values were equal according to ===.

```
cmp :: (MyEq a) => a -> a -> String
cmp t1 t2 = if t1 === t2 then "Equal" else "Not
Equal"
```

The value `result = cmp test1 test2` uses the function cmp to compare two test trees where:

```
test1 :: Tree Int
test2 :: Tree Int
```

The compiler will rewrite the cmp function and its uses. Explain how it does so.

Answer: The cmp function will be rewritten to take an extra dictionary parameter. It will use this parameter to find the appropriate definitions of the === operation. The use of cmp in result will be rewritten to pass in the dictionary inferred from the types of its arguments.

5. [4pts] Assume that the compiler generated a dictionary named `dMyEqInt` for the Int instance of `MyEq`:

```
dMyEqInt :: MyEqD Int
```

Fill in the following rewritten versions of the cmp function and result definition:

```
cmp :: MyEqD a -> a -> a -> String
cmp d t1 t2 = if (===) d t1 t2
                then "Equal" else "Not Equal"
result = cmp (dMyEqTree dMyEqInt) tree1 tree2
```

Here cmp should operate on both Trees and Ints. result, on the other hand, is defined in terms trees; the value is "Equal" if the two trees are equal according to === and "Not Equal" otherwise.

Another example

```
parabola x = (x * x) + x
parabola' (plus, times) x = plus (times x x) x
-- Dictionary type
data MathDict a = MkMathDict (a->a->a) (a->a->a)
```

This is a data declaration in Haskell, which defines a new data type called `MathDict` that is parameterized by a type variable `a`.
The `data` keyword is used to introduce a new data type definition. The name of the data type is `MathDict`, and it takes one type parameter `a`.
The = separates the type constructor (`MathDict a`) from the data constructors. In this case, there is only one data constructor, `MkMathDict`.
`MkMathDict` is the data constructor for the `MathDict` type. It takes two arguments, both of which are function types:
So, a value of type `MathDict a` is a product of two binary operations on the type `a`.

```
-- Accessor functions
get_plus :: MathDict a -> (a->a->a)
get_plus (MkMathDict p t) = p
```

get_plus takes a MathDict and returns a function that is plus

```
get_times :: MathDict a -> (a->a->a)
get_times (MkMathDict p t) = t
```

get_times takes a MathDict and returns a function that is times

```
-- "Dictionary-passing style"
parabola :: MathDict a -> a -> a
parabola dict x = let plus  = get_plus  dict
                      times = get_times dict
                  in plus (times x x) x
```

parabola now is a function that takes a MathDict and a number.

Subtyping

Interface(A)⊂Interface(B) => B <: A

$B <: A \Rightarrow \forall x|x: A, \phi(x) \rightarrow \forall y|y: B, \phi(y)$

Anywhere we can use A, we can use B

```
class Shape{}
class Circle:Shape{}
class A{
    virtual Shape create();
}
class B:A{
    virtual Shape create();
}
```

is B subtybe of A? B<:A Yes

```
void fn(A* a){
    a->create();
}
fn(new A());
fn(new B());
```

What if we change that create() function in B to be of type Circle ?
you cannot call A's version of create since A's create is virtual , and has overridden/redefined A's create , B only has one version of create
It will still work. Since in A's create it was expecting a Shape , now we are returning a Circle which is a subtype of Shape
B <: A
void -> Circle <: void -> Shape
Circle <: Shape
The above is called covariant return type
What if we swap A and B's type for create, i.e., A's create is of type Circle and B's type is Shape ?
It does not support subtyping.
Although C++ lets B to overload create (it will inherit A's create but this is not upholding subtyping)

```
class Shape{}
class Circle : Shape{}
class A{
    virtual Shape create();
    virtual void modify(Shape);
}
class B:A{
    virtual Shape create();
    virtual void modify(Circle);

void fn(A* a){
    a->create();
    a->modify(Shape);
}
fn(new A());
fn(new B());
```

B is not a subtype of A
Since A's modify can take triangle but B's can't
Covariant return type is ok but not covariant argument type
Circle->void <: Shape->void is false here.

```
class Shape{}
class Circle : Shape{}
class A{
    virtual Shape create();
    virtual void modify(Circle){};
}
class B:A{
    virtual Shape create();
    virtual void modify(Shape){};
}
void fn(A* a){
    a->create();
    a->modify(Shape);
}
```

Circle->void >: Shape->void is true. This is called contravariant arguments
B is a subtype of A.
C++ does not have contravariant arguments method specialization
C -> D <: A -> B iff A <: C and D <: B
Without virtual keyword, function will be overloaded

```
class A{ int a; void f(); }
A* pa; pa->f();
```

Compiler will rewrite `pa->f()` to `__A_f(pa)`
Virtual functions will be redefined.
ST !== Inheritance
ST ⇒Inheritance, False
Inheritance ⇒ST, False
Dynamic lookup is at runtime, haskell typeclass is done at compile time.
Typeclass is a form of overloading
DL !== (Typeclasses == Overloading)

Continuation Passing Style

Rules to do CPS

1. Every function gets a callback function cc
2. Replace all returns with the callback
3. Anytime we have a statement, then another statement, then a return

```
let result = f(x);
Statement;
return ...;
```

it gets transformed into
`f(x, result=>statement;return)`
`statement;return` will be recursively transformed

α-renaming or α-conversion
λx.e = λy.e[x:=y] where y∉FV(e)
β-reduction ➤ (λx.e1) e2 = e1 [x:=e2]
η-conversion ➤ λx.(e x) = e where x∉FV(e)