PART 2
Object-Oriented Programming

Sub Type Polymorphism (Concept)

Motivation

We want:

```
object tom {
  val name = "Tom"
  val home = "02-880-1234"
}
object bob {
  val name = "Bob"
  val mobile = "010-1111-2222"
}
def greeting(r: ???) = "Hi " + r.name + ", How are you?"
greeting(tom)
greeting(bob)
```

Note that we have

```
tom: {val name: String; val home: String}
bob: {val name: String; val mobile: String}
```

Sub Types to the Rescue!

```
import reflect.Selectable.reflectiveSelectable

type NameHome = { val name: String; val home: String }
type NameMobile = { val name: String; val mobile: String}
type Name = { val name: String }
```

`NameHome <: Name` (`NameHome` is a sub type of `Name`)

`NameMobile <: Name` (`NameMobile` is a sub type of `Name`)

```
def greeting(r: Name) = "Hi " + r.name + ", How are you?"
greeting(tom)
greeting(bob)
```

Sub Types

- The sub type relation is kind of the subset relation.
- But they are NOT the same.

- T <: S
  Every element of T can be used as that of S.

- *Cf.* T is a subset of S.
  Every element of T is that of S.

- Why polymorphism?
  A function of type S=>R can be used as T=>R for many sub types T of S.
  Note that S=>R <: T=>R when T <: S.

Summary: Subtype Polymorphism

- Subtype Polymorphism
  - Program against known datatypes with common structures
  - How is it possible?

Two Kinds of Sub Types

- Structural Sub Types (a.k.a. Duck Typing)
  - The system implicitly determines the sub type relation by the structures of data types.
  - Structurally equivalent types are treated the same.

- Nominal Sub Types (a.k.a. Ad hoc Polymorphism)
  - The user explicitly specify the sub type relation using the names of data types.
  - Structurally equivalent types with different names may be treated differently.

Structural Sub Types

General Sub Type Rules

- Reflexivity:
  For any type T, we have:

$$T <: T$$

- Transitivity:
  For any types T, S, R, we have:

$$\frac{T <: R \qquad R <: S}{T <: S}$$

Sub Types for Special Types

- Nothing: The empty set
- Any: The set of all values

- For any type T, we have:

$$\text{Nothing} <: T <: \text{Any}$$

- Example

```
val a : Int = 3
val b : Any = a
def f(a: Nothing) : Int = a
```

Sub Types for Records

- Permutation

$$\frac{}{\{\ldots;\ x\colon T1;\ y\colon T2;\ \ldots\} <: \{\ldots;\ y\colon T2,\ x\colon T1;\ \ldots\}}$$

- Width

$$\frac{}{\{\ldots;\ x\colon T;\ \ldots\} <: \{\ldots;\ \ldots\}}$$

- Depth

$$\frac{T <: S}{\{\ldots;\ x\colon T\ ;\ \ldots\} <: \{\ldots;\ x\colon S;\ \ldots\}}$$

Sub Types for Records

- Example

  {val x: { val y: Int; val z: String}, val w: Int}

  <:   (by permutation)

  {val w: Int; val x: { val y: Int; val z: String}}

  <:   (by depth & width)

  {val w: Int; val x: {val z: String}}

Sub Types for Tuples

- Depth

$$\frac{T <: S}{(\ldots, T, \ldots) <: (\ldots, S, \ldots)}$$

Sub Types for Functions

- Function Sub Type

$$\frac{T <: T' \qquad S <: S'}{(T' \Rightarrow S) <: (T \Rightarrow S')}$$

- Example

```
import reflect.Selectable.reflectiveSelectable
def foo(s: {val a: Int; val b: Int}) : {val x: Int; val y: Int} = {
  object tmp {
    val x = s.b
    val y = s.a
  }
  tmp
}
val gee: {val a: Int; val b: Int; val c: Int} => {val x: Int} =
  foo _
```

Classes

Class: Parameterized Record

```
import reflect.Selectable.reflectiveSelectable

type gee_type = {val name:String; val age: Int; def getPP(): String}
def gee_fun(_name: String, _age: Int) : gee_type = {
  if (!(_age >= 0 && _age < 200)) throw new Exception("Out of range")
  object tmp {
    val name : String = _name
    val age : Int = _age
    def getPP() : String = name + " of age " + age.toString() }
  tmp }
val gee : gee_type = gee_fun("David Jones",25)

gee.getPP()
```

Class: Parameterized Record

```
class foo_type(_name: String, _age: Int) {
  if (!(_age >= 0 && _age < 200)) throw new Exception("Out of range")
  val name : String = _name
  val age : Int = _age
  def getPP() : String = name + " of age " + age.toString() }
val foo : foo_type = new foo_type("David Jones",25)
foo.getPP()
```

use: `foo.name foo.age foo.getPP`

- `foo` is a value of `foo_type`
- `gee` is a value of `gee_type`

Class: No Structural Sub Typing

- Records: Structural sub-typing

$$\texttt{foo\_type} <: \texttt{gee\_type}$$

- Classes: Nominal sub-typing

$$\texttt{gee\_type} \not<: \texttt{foo\_type}$$

```
val v1 : gee_type = foo
val v2 : foo_type = gee  // type error

def greeting(r:{val name:String}) =
  "Hi " + r.name + ", How are you?"
greeting(foo)
```

Structural Types vs. Nominal Types

- Structural Types
  - Includes arbitrary values with the required structures as elements
  - Allows arbitrary types with the required structures as sub types
  - Cannot assume any properties on their elements

- Nominal Types
  - Includes only specific values as elements
  - Allows only specific types as sub types
  - Can assume specific properties on their elements

Class: Can be Recursive!

```
class MyList[A](v: A, nxt: Option[MyList[A]]) {
  val value : A = v
  val next : Option[MyList[A]] = nxt
}
type YourList[A] = Option[MyList[A]]

val t : YourList[Int] =
  Some(new MyList(3,Some (new MyList(4,None))))
val s : YourList[Int] =
  None
```

Note on Null value

- `null`: The special element of every class & structural type

- `null` is often used to represent None instead of using an Option type (Efficient but Not Safe)

- It is discouraged to use `null` in Scala although Scala supports `null` for compatibility with Java.

Simplification using Argument Members

```
class MyList[A](v: A, nxt: Option[MyList[A]]) {
  val value : A = v
  val next : Option[MyList[A]] = nxt
}
```

```
class MyList[A](val value:A, val next:Option[MyList[A]]) {
}
```

```
class MyList[A](val value:A, val next:Option[MyList[A]])
```

Simplification using Companion Object

```
class MyList[A](val value:A, val next:Option[MyList[A]])
object MyList
{ def apply[A](v: A, nxt: Option[MyList[A]]) =
    new MyList(v,nxt)
}

type YourList[A] = Option[MyList[A]]
object YourList
{ def apply[A](v: A, nxt: Option[MyList[A]]) =
    Some(new MyList(v,nxt))
}
val t0 = None
val t1 = Some(new MyList(3,Some(MyList(4,None))))
val t2 = YourList(3,(YourList(4,None)))
```

Nominal Sub Typing for Classes

Nominal Sub Typing, a.k.a. Inheritance

```
class foo_type(x: Int, y: Int) {
  val a : Int = x
  def b : Int = a + y
  def f(z: Int) : Int = b + y + z
}

class gee_type(x: Int) extends foo_type(x+1,x+2) {
  val c : Int = f(x) + b
}
```

gee_type <: foo_type

```
(new gee_type(30)).c
def test(f: foo_type) = f.a + f.b
test(new foo_type(10,20))
test(new gee_type(30))
```

Overriding

```
class foo_type(x: Int, y: Int) {
  val a : Int = x
  def b : Int = 0
  def f(z: Int) : Int = b * z
}
class gee_type(x: Int) extends foo_type(x+1,x+2) {
 override def b = 10
 // or, override def b = super.b + 10
 val c : Int = f(x) + b
}

(new gee_type(30)).c
def test(v: foo_type) =
 println(v.f(42))
test(new foo_type(1,2))
test(new gee_type(0))
```

Overriding vs. Overloading

```
class foo_type(x: Int, y: Int) {
  val a : Int = x
  def b : Int = 0
  def f(z: Int) : Int = b * z
}
class gee_type(x: Int) extends foo_type(x+1,x+2) {
  def f(z: String) : Int = 77
}
```

Q: Can we override with a different type?

```
override def f(z: String): Int = 77   //No,  arg: diff type
def f(z: String): Int = 77   // Overloading, arg: diff type
override def f(z: Int): Int = 77 //Yes, arg: same type
```

Example: MyList using Inheritance

```
class MyList[A](v: A, nxt: Option[MyList[A]]) {
  val value : A = v
  val next : Option[MyList[A]] = nxt
}
type YourList[A] = Option[MyList[A]]
val t : YourList[Int] =
  Some(new MyList(3,Some (new MyList(4,None))))
```

```
class MyList[A]()
class MyNil[A]() extends MyList[A]
class MyCons[A](val hd: A, val tl: MyList[A])
  extends MyList[A]
val t: MyList[Int] =
    new MyCons(3, new MyCons(4, new MyNil()))
```

Simplification: MyList

```
class MyList[A]

class MyNil[A]() extends MyList[A]
object MyNil { def apply[A]() = new MyNil[A]() }

class MyCons[A](val hd: A, val tl: MyList[A])
  extends MyList[A]
object MyCons {
  def apply[A](hd:A, tl:MyList[A]) = new MyCons[A](hd, tl)}

val t: MyList[Int] = MyCons(3, MyNil())

def length(x: MyList[Int]) = ???
```

Example: MyList with match

```
abstract class MyList[A]() {
  def matches[R](nilE: =>R, consE: (A,MyList[A]) => R) : R
}
class MyNil[A]() extends MyList[A] {
  def matches[R](nilE: =>R, consE: (A,MyList[A]) => R) : R =
    nilE
}
class MyCons[A](val hd: A, val tl: MyList[A]) extends MyList[A] {
  def matches[R](nilE: =>R, consE: (A,MyList[A]) => R) : R =
    consE(hd,tl)
}
def length[A](l: MyList[A]) : Int =
  l.matches(0,
                (hd, tl) => 1 + length(tl))

length(new MyCons(10, new MyCons(5, new MyNil())))
```

Case Class

```
sealed abstract class MyList[A] { … }
case class MyNil[A]() extends MyList[A] { … }
object MyNil { def apply[A]() = new MyNil[A]() }
case class MyCons[A](val hd: A, val tl: MyList[A])
  extends MyList[A] { … }
object MyCons {
  def apply[A](hd:A, tl:MyList[A]) = new MyCons[A](hd, tl)}
val t: MyList[Int] = MyCons(3, MyNil())
```

Allow Pattern Matching

```
def length(x: MyList[Int]): Int =
  x match {
    case MyNil() => 0
    case MyCons(hd, tl) => 1 + length(tl)
  }
```

Cf. `sealed abstract class MyList[A]`

Encoding ADT using classes: Monotonicity

```
sealed abstract class MyList[+A] {
  def matches[R](nilE: =>R, consE: (A,MyList[A])=>R) : R
  def append[B>:A](l: MyList[B]) : MyList[B]
}
object MyNil extends MyList[Nothing] {
  def matches[R](nilE: =>R, consE: (Nothing,MyList[Nothing])=>R) = nilE
  def append[B](l: MyList[B]) = l
}
class MyCons[A](val hd: A, val tl: MyList[A]) extends MyList[A] {
  def matches[R](nilE: =>R, consE: (A,MyList[A])=>R) = consE(hd,tl)
  def append[B>:A](l: MyList[B]) = new MyCons[B](hd, tl.append(l))
}
object MyCons{ def apply[A](hd:A, tl:MyList[A]) = new MyCons[A](hd, tl) }
def length[A](l: MyList[A]) : Int =
  l.matches(
    0,
    (_,tl) => 1 + length(tl) )
length(MyCons(3, MyCons(2, MyNil)).append(MyCons(1,MyNil)))
```

Abstract Classes for Interface

Abstract Class: Interface

- Abstract Classes
  - Can be used to abstract away the implementation details.

Abstract classes for Interface
Concrete sub-classes for Implementation

Abstract Class: Interface

➢ Example Interface

```
// Written by Alice
// if getValue(i) returns None, you should not use i.getNext()
abstract class Iter[A] {
  def getValue: Option[A]
  def getNext: Iter[A]
}

def sumElements[A](f: A=>Int)(xs: Iter[A]) : Int =
  xs.getValue match {
    case None => 0
    case Some(n) => f(n) + sumElements(f)(xs.getNext)
  }
def sumElementsId(xs:Iter[Int]) =
  sumElements((x:Int)=>x)(xs)
```

Concrete Class: Implementation

```scala
// Written by Bob
sealed abstract class MyList[A] extends Iter[A]
case class MyNil[A]() extends MyList[A] {
  def getValue = None
  def getNext = throw new Exception("...")
}
case class MyCons[A](hd: A, tl: MyList[A])
  extends MyList[A]
{
  def getValue = Some(hd)
  def getNext = tl
}

val t1 = MyCons(3, MyCons(5, MyCons(7, MyNil())))

sumElementsId(t1)
```

A Better Interface

```
abstract class Iter[A] {
  def get: Option[(A,Iter[A])]
}
def sumElements[A](f: A=>Int)(xs: Iter[A]) : Int =
  xs.get match {
    case None => 0
    case Some(n,nxt) => f(n) + sumElements(f)(nxt)
  }
def sumElementsId(xs:Iter[Int]) = sumElements((x:Int)=>x)(xs)
sealed abstract class MyList[A] extends Iter[A]
case class MyNil[A]() extends MyList[A] {
  def get = None }
case class MyCons[A](hd: A, tl: MyList[A]) extends MyList[A] {
  def get = Some(hd,tl) }
class IntCounter(n: Int) extends Iter[Int] {
  def get = if (n >= 0) Some(n, new IntCounter(n-1)) else None }
```

More on Abstract Classes

Problem: Iter for MyTree

```
abstract class Iter[A] {
  def getValue: Option[A]
  def getNext: Iter[A]
}

// Written by David
sealed abstract class MyTree[A]
case class Empty[A]() extends MyTree[A]
case class Node[A](value: A,
                   left: MyTree[A],
                   right: MyTree[A]) extends MyTree[A]
```

Q: Can MyTree[A] implement Iter[A]?

Try it, but it is not easy.

Possible Solution

```
// Written by David
sealed abstract class MyTree[A] extends Iter[A]
case class Empty[A]() extends MyTree[A] {
  def getValue = None
  def getNext = this  }
case class Node[A](value: A, left: MyTree[A], right: MyTree[A])
     extends MyTree[A] {
  def getValue = Some(value)
  def getNext: MyTree[A] = {
    def merge_right(l : MyTree[A]): MyTree[A] = l match {
      case Empty() => right
      case Node(v, lt, rt) => Node(v, lt, merge_right(rt))  }
    merge_right(left)  } }
val t1 = Node(3, Node(7, Node(2, Empty(), Empty()), Empty()),
                  Node(8, Empty(), Empty()))
sumElements[Int]((x)=>x*x)(t1)
```

Solution: Better Interface

```
abstract class Iter[A] {
  def getValue: Option[A]
  def getNext: Iter[A]
}
abstract class Iterable[A] {
  def iter : Iter[A]
}

def sumElements[A](f: A=>Int)(xs: Iter[A]) : Int =
  xs.getValue match {
    case None => 0
    case Some(n) => f(n) + sumElements(f)(xs.getNext)
  }
def sumElementsGen[A](f: A=>Int)(xs: Iterable[A]) : Int =
  sumElements(f)(xs.iter)
```

Let's Use MyList

```
sealed abstract class MyList[A] extends Iter[A]
case class MyNil[A]() extends MyList[A] {
  def getValue = None
  def getNext = throw new Exception("...")
}
case class MyCons[A](val hd: A, val tl: MyList[A])
  extends MyList[A] {
  def getValue = Some(hd)
  def getNext = tl
}
```

MyTree <: Iterable (Try)

```scala
sealed abstract class MyTree[A] extends Iterable[A]

case class Empty[A]() extends MyTree[A] {
  val iter = MyNil()
}

case class Node[A](value: A,
                   left: MyTree[A],
                   right: MyTree[A]) extends MyTree[A] {
  // "val iter" is more specific than "def iter",
  // so it can be used in a sub type.
  // In this example, "val iter" is also
  // more efficient than "def iter".
  val iter = MyCons(value, ???(left.iter,right.iter))
}
```

Extend MyList with append

```
sealed abstract class MyList[A] extends Iter[A] {
  def append(lst: MyList[A]) : MyList[A]
}
case class MyNil[A]() extends MyList[A] {
  def getValue = None
  def getNext = throw new Exception("...")
  def append(lst: MyList[A]) = lst
}
case class MyCons[A](val hd: A, val tl: MyList[A])
  extends MyList[A]
{
  def getValue = Some(hd)
  def getNext = tl
  def append(lst: MyList[A]) = MyCons(hd,tl.append(lst))
}
```

MyTree <: Iterable

```
sealed abstract class MyTree[A] extends Iterable[A] {
  def iter : MyList[A]
  // Note:
  // def iter : Int // Type Error because not (Int <: Iter[A])
}
case class Empty[A]() extends MyTree[A] {
  val iter = MyNil()
}
case class Node[A](value: A,
                   left: MyTree[A],
                   right: MyTree[A]) extends MyTree[A] {
  def iter = MyCons(value, left.iter.append(right.iter))
  // def iter = left.iter.append(MyCons(value,right.iter))
  // def iter = left.iter.append(right.iter.append(
  //                    MyCons(value,MyNil())))
}
```

Test

```
def generateTree(n: Int) : MyTree[Int] = {
  def gen(lo:Int, hi: Int) : MyTree[Int] =
    if (lo > hi) Empty()
    else {
      val mid = (lo+hi)/2
      Node(mid, gen(lo,mid-1), gen(mid+1,hi))
    }
  gen(1,n)
}

sumElementsGen((x:Int)=>x)(generateTree(100))
```

Iter <: Iterable

```
abstract class Iterable[A] {
  def iter : Iter[A]
}

abstract class Iter[A] extends Iterable[A] {
  def getValue: Option[A]
  def getNext: Iter[A]
  def iter = this
}

val lst : MyList[Int] =
  MyCons(3, MyCons(4, MyCons(2,MyNil())))

sumElementsGen ((x:Int)=>x)(lst)
```

Note: tail-recursive "append"

```
sealed abstract class MyList[A] extends Iter[A] {
  def append(lst: MyList[A]) : MyList[A] =
    MyList.revAppend(MyList.revAppend(this, MyNil()), lst)
}
object MyList { //  Mutual references are allowed between class T and object T
  //  Tail-recursive functions should be written in "object", or as final methods
  def revAppend[A](lst1: MyList[A], lst2: MyList[A]): MyList[A] =
    lst1 match {
      case MyNil() => lst2
      case MyCons(hd, tl) => revAppend(tl, MyCons(hd, lst2))
    }
}
case class MyNil[A]() extends MyList[A] {
  def getValue = None
  def getNext = throw new Exception("...") }
case class MyCons[A](val hd:A, val tl:MyList[A]) extends MyList[A] {
  def getValue = Some(hd)
  def getNext = tl }
```

Lazy List

Problem: Inefficiency

```
def time[R](block: => R): R = {
  val t0 = System.nanoTime()
  val result = block    // call-by-name
  val t1 = System.nanoTime()
  println("Elapsed time: " + ((t1 - t0)/1000000) + "ms"); result
}
def sumN[A](f: A=>Int)(n: Int, xs: Iterable[A]) : Int = {
  def sumIter(res : Int, n: Int, xs: Iter[A]) : Int =
    if (n <= 0) res
    else xs.getValue match {
      case None => res
      case Some(v) => sumIter(f(v) + res, n-1, xs.getNext)
    }
  sumIter(0,n,xs.iter)
}
// Problem: takes a few seconds to get a single value
{ val t: MyTree[Int] = generateTree(200000)
  time (sumN((x:Int) => x)(1, t)) }
```

Solution 1: Using Lists of Trees

```
class MyTreeIter[A](val lst: MyList[MyTree[A]]) extends Iter[A] {
 val getValue = lst match {
   case MyCons(Node(v,_,_), _) => Some(v)
   case _ => None
 }
 def getNext = {
   val remainingTrees : MyList[MyTree[A]] = lst match {
     case MyNil() => throw new Exception("...")
     case MyCons(hd,tl) => hd match {
       case Empty() => throw new Exception("...")
       case Node(_,Empty(),Empty()) => tl
       case Node(_,lt,Empty()) => MyCons(lt,tl)
       case Node(_,Empty(),rt) => MyCons(rt,tl)
       case Node(_,lt,rt) => MyCons(lt,MyCons(rt,tl))
     }
   }
   new MyTreeIter(remainingTrees)
 } }
```

Lazy Iteration using Lists of Trees

```
sealed abstract class MyTree[A] extends Iterable[A]
case class Empty[A]() extends MyTree[A] {
  val iter = new MyTreeIter(MyNil())
}
case class Node[A](value: A,
                   left: MyTree[A],
                   right: MyTree[A]) extends MyTree[A]
{
  val iter = new MyTreeIter(MyCons(this,MyNil()))
}

{ val t: MyTree[Int] = generateTree(200000)
  time (sumN((x:Int) => x)(100, t))
  time (sumN((x:Int) => x)(100000, t))
}
```

Solution 2: Lazy List

```
sealed abstract class LazyList[A] extends Iter[A] {
  def append(lst: LazyList[A]) : LazyList[A]
}

case class LNil[A]() extends LazyList[A] {
  def getValue = None
  def getNext = throw new Exception("")
  def append(lst: LazyList[A]) = lst
}

class LCons[A](hd: A, _tl: =>LazyList[A]) extends LazyList[A] {
  lazy val tl = _tl
  def getValue = Some(hd)
  def getNext = tl
  def append(lst: LazyList[A]) = LCons(hd, tl.append(lst)) }
object LCons {
  def apply[A](hd: A, tl: =>LazyList[A]) = new LCons(hd, tl)
}
```

Note: "append" is not recursive!!!

Lazy Iteration using LazyList

```
sealed abstract class MyTree[A] extends Iterable[A] {
  def iter : LazyList[A]
}
case class Empty[A]() extends MyTree[A] {
  val iter = LNil()
}
case class Node[A](value: A,
                   left: MyTree[A],
                   right: MyTree[A]) extends MyTree[A] {
  lazy val iter = LCons(value, left.iter.append(right.iter))
  // lazy val iter = left.iter.append(LCons(value,right.iter))
  // lazy val iter = left.iter.append(right.iter.append(
  //                    LCons(value,LNil())))
}
{ val t: MyTree[Int] = generateTree(200000)
  time (sumN((x:Int) => x)(100, t))
  time (sumN((x:Int) => x)(100000, t))
}
```

Note: "iter" is not recursive!!!

Wrapper for Inheritance

Using a Wrapper Class

```
abstract class Iter[A] {
  def getValue: Option[A]
  def getNext: Iter[A]
}

class ListIter[A](val list: List[A]) extends Iter[A] {
  def getValue = list.headOption
  def getNext = new ListIter(list.tail)
}

sumElements((x:Int)=>x)(new ListIter(List(1,2,3,4)))
```

MyTree Using ListIter

```
abstract class Iterable[A] {
  def iter : Iter[A]
}
sealed abstract class MyTree[A] extends Iterable[A] {
  def iter : ListIter[A]
}
case class Empty[A]() extends MyTree[A] {
  val iter : ListIter[A] = new ListIter(Nil)
}
case class Node[A](value: A,
                   left: MyTree[A],
                   right: MyTree[A])
  extends MyTree[A] {
  val iter : ListIter[A] = new ListIter(
    value::(left.iter.list ++ right.iter.list))
}
```

Test

```
val t : MyTree[Int] =
  Node(3, Node(4,Node(2,Empty(),Empty()),
    Node(3,Empty(),Empty())),
    Node(5,Empty(),Empty()))

sumElementsGen((x:Int)=>x)(t)
```

Abstract Class With Associate Types

Using an Associate Type

```
abstract class Iterable[A] {
  type iter_t
  def iter: iter_t
  def getValue(i: iter_t) : Option[A]
  def getNext(i: iter_t) : iter_t
}

def sumElements[A](f:A=>Int)(xs: Iterable[A]) : Int = {
  def sumElementsIter(i: xs.iter_t) : Int =
    xs.getValue(i) match {
      case None => 0
      case Some(n) => f(n) + sumElementsIter(xs.getNext(i))
    }
  sumElementsIter(xs.iter)
}
```

MyTree Using List

```
sealed abstract class MyTree[A] extends Iterable[A] {
  type iter_t = List[A]
  def getValue(i: List[A]): Option[A] = i.headOption
  def getNext(i: List[A]): List[A] = i.tail
}
case class Empty[A]() extends MyTree[A] {
  val iter : List[A] = Nil
}
case class Node[A](value: A,
                   left: MyTree[A], right: MyTree[A])
  extends MyTree[A] {
  val iter = value :: (left.iter ++ right.iter) //Pre-order
//val iter = left.iter ++ (value :: right.iter) // In-order
//val iter = left.iter ++ (right.iter ++ List(value))
                                             //Post-order
}
```

Test

```
val t : MyTree[Int] =
  Node(3, Node(4,Node(2,Empty(),Empty()),
    Node(3,Empty(),Empty())),
    Node(5,Empty(),Empty()))

sumElements((x:Int)=>x)(t)
```

Abstract Class with Arguments

Abstract Class with Arguments

```
abstract class IterableH[A] extends Iterable[A] {
 def hasElement(a: A) : Boolean
}
abstract class IterableHE[A](eq: (A,A) => Boolean)
  extends IterableH[A]
{
  def hasElement(a: A) : Boolean = {
    def hasElementIter(i: iter_t) : Boolean =
      getValue(i) match {
        case None => false
        case Some(n) =>
          if (eq(a,n)) true
          else hasElementIter(getNext(i))
      }
    hasElementIter(iter)
  }
}
```

MyTree

```
sealed abstract class MyTree[A](eq:(A,A)=>Boolean)
  extends IterableHE[A](eq) {
  type iter_t = List[A]
  def getValue(i : List[A]) : Option[A] = i.headOption
  def getNext(i: List[A]) : List[A] = i.tail
}
case class Empty[A](eq: (A,A)=>Boolean)
  extends MyTree[A](eq) {
  val iter : List[A] = Nil
}
case class Node[A](eq: (A,A)=>Boolean,
                   value: A, left: MyTree[A], right: MyTree[A])
  extends MyTree[A](eq) {
  val iter : List[A] = value :: (left.iter ++ right.iter)
}
```

Test

```
val leq = (x:Int,y:Int) => x == y
val lEmpty = Empty(leq)
def lNode(n: Int, t1: MyTree[Int], t2: MyTree[Int]) =
  Node(leq,n,t1,t2)

val t : MyTree[Int] =
  lNode(3, lNode(4,lNode(2,lEmpty,lEmpty),
                   lNode(3,lEmpty,lEmpty)),
           lNode(5,lEmpty,lEmpty))

sumElements((x:Int)=>x)(t)
t.hasElement(5)
t.hasElement(10)
```

Alternatively, Argument Elimination

```
abstract class IterableHE[A]
  extends Iterable[A]
{
  def eq(a:A, b:A) : Boolean
  def hasElement(a: A) : Boolean = {
    def hasElementIter(i: iter_t) : Boolean =
      getValue(i) match {
        case None => false
        case Some(n) =>
          if (eq(a,n)) true
          else hasElementIter(getNext(i))
      }
    hasElementIter(iter)
  }
}
```

MyTree

```
sealed abstract class MyTree[A] extends IterableHE[A] {
  type iter_t = List[A]
  def getValue(i : List[A]) : Option[A] = i.headOption
  def getNext(i: List[A]) : List[A] = i.tail
}
case class Empty[A](_eq:(A,A)=>Boolean) extends MyTree[A] {
  def eq(a:A, b:A) = _eq(a,b)
  val iter : List[A] = Nil
}
case class Node[A](_eq: (A,A)=>Boolean,
                   value: A, left: MyTree[A], right: MyTree[A])
  extends MyTree[A] {
  def eq(a:A, b:A) = _eq(a,b)
  val iter : List[A] = value :: (left.iter ++ right.iter)
}
```

Test

```
val leq = (x:Int,y:Int) => x == y
val IEmpty = Empty(leq)
def INode(n: Int, t1: MyTree[Int], t2: MyTree[Int]) =
  Node(leq,n,t1,t2)

val t : MyTree[Int] =
  INode(3, INode(4,INode(2,IEmpty,IEmpty),
                   INode(3,IEmpty,IEmpty)),
           INode(5,IEmpty,IEmpty))

sumElements((x:Int)=>x)(t)
t.hasElement(5)
t.hasElement(10)
```

More on Classes

Motivating Example

```scala
class Primes(val prime: Int, val primes: List[Int]) {
  def getNext: Primes = {
    val p = computeNextPrime(prime + 2)
    new Primes(p, primes ++ (p :: Nil))
  }
  def computeNextPrime(n: Int) : Int =
    if (primes.forall((p:Int) => n%p != 0)) n
    else computeNextPrime(n+2)
}

def nthPrime(n: Int): Int = {
  def go(primes: Primes, k: Int): Int =
    if (k <= 1) primes.prime
    else go(primes.getNext, k - 1)
  if (n <= 0) 2 else go(new Primes(3, List(3)), n)
}
nthPrime(10000)
```

Multiple Constructors

```
class Primes(val prime: Int, val primes: List[Int]) {
  def this() = this(3, List(3))
  def getNext: Primes = {
    val p = computeNextPrime(prime + 2)
    new Primes(p, primes ++ (p :: Nil))
  }
  def computeNextPrime(n: Int) : Int =
    if (primes.forall((p:Int) => n%p != 0)) n
    else computeNextPrime(n+2)
}

def nthPrime(n: Int): Int = {
  def go(primes: Primes, k: Int): Int =
    if (k <= 1) primes.prime
    else go(primes.getNext, k - 1)
  if (n == 0) 2 else go(new Primes, n)
}
nthPrime(10000)
```

Access Modifiers

- Access Modifiers
  - Private: Only the class can access the member.
  - Protected: Only the class and its sub classes can access the member.

Using Access Modifiers

```
class Primes private (val prime:Int, protected val primes:List[Int])
{ def this() = this(3, List(3))
  def getNext: Primes = {
    val p = computeNextPrime(prime + 2)
    new Primes(p, primes ++ (p :: Nil))
  }
  private def computeNextPrime(n: Int) : Int =
    if (primes.forall((p:Int) => n%p != 0)) n
    else computeNextPrime(n+2)
}

def nthPrime(n: Int): Int = {
  def go(primes: Primes, k: Int): Int =
    if (k <= 1) primes.prime
    else go(primes.getNext, k - 1)
  if (n == 0) 2 else go(new Primes, n)
}
nthPrime(10000)
```

Traits for Multiple Inheritance

Multiple Inheritance Problem

- Multiple Inheritance
  - The famous “diamond problem”

```
class A(val a: Int)
class B extends A(10)
class C extends A(20)
class D extends B, C.
```

Problem 1: What is the value of (new D).a ?

Problem 2: The constructor of A must be executed once because A may contain side effects such as sending messages over the network.

Scala's Solution: Trait

- Traits
  - A trait can implement any of its methods,
    but should have only one constructor with no arguments.
  - An [abstract] class (resp. trait) X can
    "extends" one trait or [abstract] class with any (resp. no) arguments
    "with" multiple traits T_1 , … , T_n
    such that, for each i,
    the least superclass of T_i, if exists, should be a superclass of X
    where
    C is a superclass of T if C is an (abstract) class and T transitively "extends" C.
  - No cyclic inheritance is allowed.
- Property
  - For any ancestor class in the inheritance tree of a class:
    - Its constructor with arguments can appear at most once
    - Its constructor with no argument can appear multiple times

Example

```
class A(val a : Int) {
  def this () = this(0)
}
trait B {
  def f(x: Int): Int = x
}
trait C extends A with B {
  def g(x: Int): Int = x + a
}
trait D extends B {
  def h(x: Int): Int = f(x + 50)
}
class E extends A(10) with C with D {
  override def f(x: Int) = x * a
}

val e = new E
```

Algorithm for Multiple Inheritance

- Algorithm
  - Give a linear order among all ancestors by "post-order" traversing without revisiting the same node.
  - Invoke the constructors once in that order.

    Note. Post-order traversal of a class C means
    - Recursively post-order traverse C's first parent; …;
    - Recursively post-order traverse C's last parent; and
    - Visit C.

    By post-order traversing from "E" in the previous example, we have the order: `A(10) -> B -> C -> D -> E`

```
val e = new E
e.a // 10                    e.f(100) // 100*10
e.g(100) // 100 + 10         e.h(100) // (100 + 50) * 10
```

  - A constructor with arguments is always visited before the same constructor with no arguments.
  - Compile error if the same field is implemented by multiple classes

A Simple Example With Traits

Motivation

```
abstract class Iter[A] {
  def getValue: Option[A]
  def getNext: Iter[A]
}

class ListIter[A](val list: List[A]) extends Iter[A] {
  def getValue = list.headOption
  def getNext = new ListIter(list.tail)
}

abstract class Dict[K,V] {
  def add(k: K, v: V): Dict[K,V]
  def find(k: K): Option[V]
}
```

Q: How can we extend ListIter and implement Dict?

Interface using Traits

```
// abstract class Dict[K,V] {
//   def add(k: K, v: V): Dict[K,V]
//   def find(k: K): Option[V] }

trait Dict[K,V] {
  def add(k: K, v: V): Dict[K,V]
  def find(k: K): Option[V]
}
```

Implementing Traits

```
class ListIterDict[K,V]
      (eq: (K,K)=>Boolean, list: List[(K,V)])
      extends ListIter[(K,V)](list)
         with Dict[K,V]
{
  def add(k:K,v:V): ListIterDict[K,V] =
    new ListIterDict(eq,(k,v)::list)
  def find(k: K) : Option[V] = {
    def go(l: List[(K, V)]): Option[V] = l match {
        case Nil => None
        case (k1, v1) :: tl =>
          if (eq(k, k1)) Some(v1) else go(tl) }
    go(list) }
}
```

Test

```
def sumElements[A](f: A=>Int)(xs: Iter[A]) : Int =
  xs.getValue match {
    case None => 0
    case Some(n) => f(n) + sumElements(f)(xs.getNext)
  }

def find3(d: Dict[Int,String]) = {
  d.find(3)
}

val d0 = new ListIterDict[Int,String]((x,y)=>x==y,Nil)
val d = d0.add(4,"four").add(3,"three")

sumElements[(Int,String)](x=>x._1)(d)
find3(d)
```

Mixin with Traits

Motivation: Mixin Functionality

```
abstract class Iter[A] {
  def getValue: Option[A]
  def getNext: Iter[A]
}

class ListIter[A](val list: List[A]) extends Iter[A]
{
  def getValue = list.headOption
  def getNext: ListIter[A] = new ListIter(list.tail)
}

trait MRIter[A] extends Iter[A] {
  def mapReduce[B,C](combine: (B,C)=>C, ival: C, f: A=>B): C = ???
}
```

Mixin Composition

```
trait MRIter[A] extends Iter[A] {
  override def getNext: MRIter[A]
  def mapReduce[B,C](combine: (B,C)=>C, ival: C, f: A=>B): C =
    getValue match {
      case None => ival
      case Some(v) =>
        combine(f(v), getNext.mapReduce(combine, ival, f))
    }
}

class MRListIter[A](list: List[A])
  extends ListIter (list) with MRIter[A]
{
 override def getNext = new MRListIter(super.getNext.list)
                        // new MRListIter(list.tail)
}
val mr = new MRListIter[Int](List(3,4,5))
mr.mapReduce[Int,Int]((b,c)=>b+c,0,(a)=>a*a)
```

Mixin Composition: A Better Way

```
trait MRIter[A] extends Iter[A] {
  def mapReduce[B,C](combine: (B,C)=>C, ival: C, f: A=>B): C = {
    def loop(c: Iter[A]): C = c.getValue match {
      case None => ival
      case Some(v) => combine(f(v), loop(c.getNext))
    }
    loop(this)
  }
}

class MRListIter[A](list: List[A])
  extends ListIter (list) with MRIter[A]

val mr = new MRListIter[Int](List(3,4,5))

// or, val mr = new ListIter(List(3,4,5)) with MRIter[Int]

mr.mapReduce[Int,Int]((b,c)=>b+c,0,(a)=>a*a)
```

Syntactic Sugar: new A with B with C { … }

```
new A(...) with B1 ··· with Bm {
  code
}
```

is equivalent to

```
{
  class _tmp_(args) extends A(args) with B1 ··· with Bm {
    code
  }
  new _tmp_(...)
}
```

Intersection Types

Intersection Types

- Typing Rule

$$\frac{t : T1 \qquad t : T2}{t : T1 \text{ with } T2}$$

- Example

```scala
trait A { val a: Int = 0 }
trait B { val b: Int = 0 }
class C extends A with B {
  override val a = 10
  override val b = 20
  val c = 30
}
val x = new C
val y: A with B = x
y.a // 10
y.b // 20
y.c // type error
```

Subtype Relation for “with”

The subtype relation for “with” is structural.

- Permutation

$$\frac{}{\ldots \text{ with } T1 \text{ with } T2 \ldots <: \ldots \text{ with } T2 \text{ with } T1 \ldots}$$

- Width

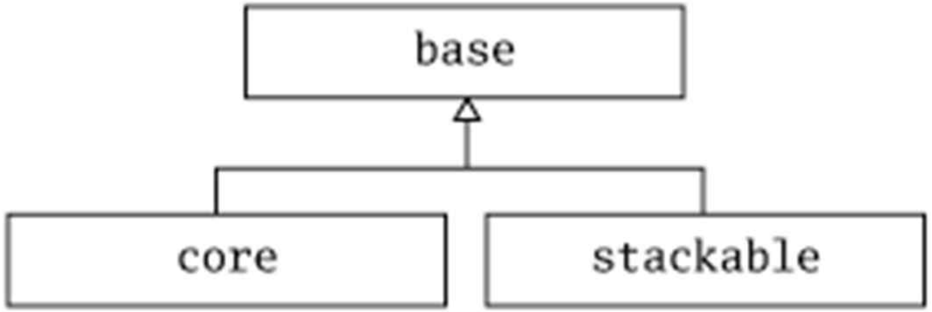
$$\frac{}{\ldots \text{ with } T \ldots <: \ldots \ldots}$$

- Depth

$$\frac{T <: S}{\ldots \text{ with } T \ldots <: \ldots \text{ with } S \ldots}$$

Stacking with Traits

Typical Hierarchy in Scala



- **BASE**
  Interface (trait or abstract class)

- **CORE**
  Functionality (trait or concrete class)

- **CUSTOM**
  Modifications (each in a separate, composable trait)

IntStack: Base

- **BASE**

```
trait Stack[A] {
  def get(): (A, Stack[A])
  def put(x: A): Stack[A]
}
```

IntStack: Core

➢**CORE**

```
class BasicIntStack protected (xs: List[Int]) extends Stack[Int]
{
  override val toString = "Stack:" + xs.toString
  def this() = this(Nil)

  def get():(Int,Stack[Int]) = (xs.head,new BasicIntStack(xs.tail))
  def put(x:Int): Stack[Int] = new BasicIntStack(x :: xs)
}

val s0 = new BasicIntStack
val s1 = s0.put(3)
val s2 = s1.put(-2)
val s3 = s2.put(4)
val (v1,s4) = s3.get()
val (v2,s5) = s4.get()
```

IntStack: Custom Modifications

- **CUSOM**

```
trait Doubling extends Stack[Int] {
  abstract override def put(x: Int): Stack[Int] = super.put(2 * x)
}

trait Incrementing extends Stack[Int] {
  abstract override def put(x: Int): Stack[Int] = super.put(x + 1)
}

trait Filtering extends Stack[Int] {
  abstract override def put(x: Int): Stack[Int] =
    if (x >= 0) super.put(x) else this
}
```

IntStack: Stacking

- **Stacking**

```
class DIFIntStack protected (xs: List[Int])
  extends BasicIntStack(xs)
  with Doubling with Incrementing with Filtering
{
  def this() = this(Nil)
}

val s0 = new DIFIntStack
val s1 = s0.put(3)
val s2 = s1.put(-2)
val s3 = s2.put(4)
val (v1,s4) = s3.get()
val (v2,s5) = s4.get()
val (v2,s6) = s5.get()
```

IntStack: Core (Correct)

➢**CORE**

```
class BasicIntStack protected (xs: List[Int]) extends Stack[Int]
{
  override val toString = "Stack:" + xs.toString
  def this() = this(Nil)
  protected def mkStack(xs: List[Int]): Stack[Int] =
    new BasicIntStack(xs)
  def get(): (Int,Stack[Int]) = (xs.head, mkStack(xs.tail))
  def put(x: Int): Stack[Int] = mkStack(x :: xs)
}

val s0 = new BasicIntStack
val s1 = s0.put(3)
val s2 = s1.put(-2)
val s3 = s2.put(4)
val (v1,s4) = s3.get()
val (v2,s5) = s4.get()
```

IntStack: Stacking (Correct)

- **Stacking**

```
class DIFIntStack protected (xs: List[Int])
  extends BasicIntStack(xs)
  with Doubling with Incrementing with Filtering
{
  def this() = this(Nil)
  override def mkStack(xs: List[Int]): Stack[Int] =
    new DIFIntStack(xs)
}

val s0 = new DIFIntStack
val s1 = s0.put(3)
val s2 = s1.put(-2)
val s3 = s2.put(4)
val (v1,s4) = s3.get()
val (v2,s5) = s4.get()
```

PART 3

Type Classes for Interfaces

Problems with OOP

Subtype Polymorphism

```
trait Ord {
  // this cmp that < 0  iff this < that
  // this cmp that > 0  iff this > that
  // this cmp that == 0 iff this == that
  def cmp(that: Ord): Int

  def ===(that: Ord): Boolean = (this.cmp(that)) == 0
  def <  (that: Ord): Boolean = (this cmp that) < 0
  def >  (that: Ord): Boolean = (this cmp that) > 0
  def <= (that: Ord): Boolean = (this cmp that) <= 0
  def >= (that: Ord): Boolean = (this cmp that) >= 0
}
def max3(a: Ord, b: Ord, c: Ord) : Ord =
  if (a <= b) { if (b <= c) c else b }
  else        { if (a <= c) c else a }
```

* Problem: hard (almost impossible) to implement Ord (e.g., using Int)

Interface over Parameter Types

```
trait Ord[A] {
  def cmp(that: A): Int

  def ===(that: A): Boolean = (this.cmp(that)) == 0
  def <  (that: A): Boolean = (this cmp that) < 0
  def >  (that: A): Boolean = (this cmp that) > 0
  def <= (that: A): Boolean = (this cmp that) <= 0
  def >= (that: A): Boolean = (this cmp that) >= 0
}
def max3[A <: Ord[A]](a: A, b: A, c: A) : A =
  if (a <= b) {if (b <= c) c else b }
  else        {if (a <= c) c else a }

class OInt(val value : Int) extends Ord[OInt] {
  def cmp(that: OInt) = value - that.value
}
max3(new OInt(3), new OInt(2), new OInt(10)).value
```

Further example: Ordered Bag

```
class Bag[U <: Ord[U]] protected (val toList: List[U]) {
  def this() = this(Nil)
  def add(x: U) : Bag[U] = {
    def go(elmts: List[U]): List[U] =
      elmts match {
        case Nil => x :: Nil
        case e :: _ if (x < e) => x :: elmts
        case e :: _ if (x === e) => elmts
        case e :: rest => e :: go(rest)
      }
    new Bag(go(toList))
  }
}
val emp = new Bag[OInt]()
val b = emp.add(new OInt(3)).add(new OInt(2)).
            add(new OInt(10)).add(new OInt(2))
b.toList.map((x)=>x.value)
```

Problems with OOP

1. Needs “subtyping” like “OInt <: Ord[OInt]”, which is quite complex as we have seen (and moreover, involves more complex concepts like variance).

2. Needs a wrapper class like “OInt” in order to add a new interface to an existing type like “Int”.

3. Interface only contains only “elimination” functions, not “introduction” functions.

4. No canonical operator

5. …

Type Classes

Separating Functions from Data

```
trait Ord[A] {
  def cmp(self: A)(a: A): Int

  def ===(self: A)(a: A) = cmp(self)(a) == 0
  def <  (self: A)(a: A) = cmp(self)(a) < 0
  def >  (self: A)(a: A) = cmp(self)(a) > 0
  def <= (self: A)(a: A) = cmp(self)(a) <= 0
  def >= (self: A)(a: A) = cmp(self)(a) >= 0
}

def max3[A](a: A, b: A, c: A)(implicit ORD: Ord[A]) : A =
  if (ORD.<=(a)(b)) {if (ORD.<=(b)(c)) c else b }
  else              {if (ORD.<=(a)(c)) c else a }

// behaves like Int <: Ord in OOP
implicit val intOrd : Ord[Int] = new {
  def cmp(self: Int)(a: Int) = self - a }
max3(3,2,10) // 10
```

Implicit

- Implicit
  - An argument is given “implicitly”

```scala
def foo(s: String)(implicit t: String) = s + t

implicit val exclamation : String = "!!!!!!"

foo("Hi")
foo("Hi")("???") // can give it explicitly
```

Syntax for type class: syntactic sugar

```
trait Ord[A]:
  extension (self: A)
    def cmp(a: A): Int
    def ===(a: A) = self.cmp(a) == 0
    def <  (a: A) = self.cmp(a) < 0
    def >  (a: A) = self.cmp(a) > 0
    def <= (a: A) = self.cmp(a) <= 0
    def >= (a: A) = self.cmp(a) >= 0

def max3[A: Ord](a: A, b: A, c: A) : A =
  if (a <= b) { if (b <= c) c else b }
  else        { if (a <= c) c else a }

given intOrd : Ord[Int] with
  extension (self: Int)
    def cmp(a: Int) = self - a


max3(3,2,10) // 10
```

Syntax for type class: syntactic sugar

```
trait Ord[A]:

  def cmp(self: A)(a: A): Int
  def ===(self: A)(a: A) = cmp(self)(a) == 0
  def < (self: A)(a: A) = cmp(self)(a) < 0
  def > (self: A)(a: A) = cmp(self)(a) > 0
  def <= (self: A)(a: A) = cmp(self)(a) <= 0
  def >= (self: A)(a: A) = cmp(self)(a) >= 0

def max3[A](a: A, b: A, c: A)(implicit ORD: Ord[A]) : A =
  if (ORD.<=(a)(b)) { if (ORD.<=(b)(c)) c else b }
  else              { if (ORD.<=(a)(c)) c else a }

implicit def intOrd : Ord[Int] = new {
  def cmp(self:Int)(a: Int) = self - a
}


max3(3,2,10) // 10
```

Bag Example using type class

```
class Bag[A: Ord] protected (val toList: List[A])
{ def this() = this(Nil)
  def add(x: A) : Bag[A] = {
    def loop(elmts: List[A]) : List[A] =
      elmts match {
        case Nil => x :: Nil
        case e :: _ if (x < e) => x :: elmts
        case e :: _ if (x === e) => elmts
        case e :: rest => e :: loop(rest)
      }
    new Bag(loop(toList))
  }
}

(new Bag[Int]()).add(3).add(2).add(3).add(10).toList
```

Bag Example using type class

```
class Bag[A] protected (val toList: List[A])(implicit ORD: Ord[A])
{ def this()(implicit ORD: Ord[A]) = this(Nil)
  def add(x: A) : Bag[A] = {
    def loop(elmts: List[A]) : List[A] =
      elmts match {
        case Nil => x :: Nil
        case e :: _ if (ORD.<(x)(e)) => x :: elmts
        case e :: _ if (ORD.===(x)(e)) => elmts
        case e :: rest => e :: loop(rest)
      }
    new Bag(loop(toList))
  }
}

(new Bag[Int]()).add(3).add(2).add(3).add(10).toList
```

Bootstrapping Implicits

```scala
// lexicographic order
given tupOrd[A, B](using Ord[A], Ord[B]): Ord[(A,B)] with
  extension (self: (A,B))
    def cmp(a: (A, B)) : Int = {
      val c1 = self._1.cmp(a._1)
      if (c1 != 0) c1
      else { self._2.cmp(a._2) }
    }

val b = new Bag[(Int,(Int,Int))]
b.add((3,(3,4))).add((3,(2,7))).add((4,(0,0))).toList
```

Bootstrapping Implicits

```
// lexicographic order
implicit def tupOrd[A, B](implicit ORDA: Ord[A], ORDB: Ord[B]): Ord[(A,B)] =
new {
  def cmp(self:(A,B))(a: (A, B)) : Int = {
    val c1 = ORDA.cmp(self._1)(a._1)
    if (c1 != 0) c1
    else { ORDB.cmp(self._2)(a._2) }
  }
}

val b = new Bag[(Int,(Int,Int))]
b.add((3,(3,4))).add((3,(2,7))).add((4,(0,0))).toList
```

With Different Orders

```
def intOrdRev : Ord[Int] = new {
  extension (self: Int)
    def cmp(a: Int) = a - self
}

(new Bag[Int]()).add(3).add(2).add(10).toList
(new Bag[Int]()(intOrdRev)).add(3).add(2).add(10).toList
```

With Different Orders

```
def intOrdRev : Ord[Int] = new {
  def cmp(self: Int)(a: Int) = a - self
}

(new Bag[Int]()).add(3).add(2).add(10).toList
(new Bag[Int]()(intOrdRev)).add(3).add(2).add(10).toList
```

Type Classes: Abstraction

Interfaces I: elimination

```
trait Iter[I,A]:
  extension (self: I)
    def getValue: Option[A]
    def getNext: I

trait Iterable[I,A]:
  type Itr
  given ITR: Iter[Itr,A]
  extension (self: I)
    def iter: Itr

// behaves like Iter[A] <: Iterable[A] in OOP
given iter2iterable[I,A](using _ITR: Iter[I,A]): Iterable[I,A] with
  type Itr = I
  def ITR = _ITR
  extension (self: I)
    def iter = self
```

Interfaces I: elimination

```
trait Iter[I,A]:
  def getValue(self: I): Option[A]
  def getNext(self: I): I

trait Iterable[I,A]:
  type Itr
  implicit def ITR: Iter[Itr,A]
  def iter(self: I): Itr

// behaves like Iter[A] <: Iterable[A] in OOP
implicit def iter2iterable[I,A](implicit _ITR: Iter[I,A]): Iterable[I,A] = new {
  type Itr = I
  def ITR = _ITR
  def iter(self: I) = self
}
```

Programs for Testing: use Iter, Iterable

```
def sumElements[I](xs: I)(implicit ITRA:Iterable[I,Int]) = {
  def loop(i: ITRA.Itr): Int =
    i.getValue match {
      case None => 0
      case Some(n) => n + loop(i.getNext)
    }
  loop(xs.iter)
}

def printElements[I,A](xs: I)(implicit ITRA: Iterable[I,A]) = {
  def loop(i: ITRA.Itr): Unit =
    i.getValue match {
      case None =>
      case Some(a) => {println(a); loop(i.getNext)}
    }
  loop(xs.iter)
}
```

Programs for Testing: use Iter, Iterable

```
def sumElements[I](xs: I)(implicit ITRA:Iterable[I,Int]) = {
  def loop(i: ITRA.Itr): Int =
    ITRA.ITR.getValue(i) match {
      case None => 0
      case Some(n) => n + loop(ITRA.ITR.getNext(i))
    }
  loop(ITRA.iter(xs))
}

def printElements[I,A](xs: I)(implicit ITRA: Iterable[I,A]) = {
  def loop(i: ITRA.Itr): Unit =
    ITRA.ITR.getValue(i) match {
      case None =>
      case Some(a) => {println(a); loop(ITRA.ITR.getNext(i))}
    }
  loop(ITRA.iter(xs))
}
```

Interfaces II: introduction + elimination

```
trait Listlike[L,A]:
  extension(u:Unit)
    def unary_! : L
  extension(elem:A)
    def ::(l: =>L): L
  extension(l: L)
    def head: Option[A]
    def tail: L
    def ++(l2: L): L

trait Treelike[T,A]:
  extension(u:Unit)
    def unary_! : T
  extension(a:A)
    def has(lt: T, rt: T): T
  extension(t: T)
    def root : Option[A]
    def left : T
    def right : T
```

Interfaces II: introduction + elimination

```
trait Listlike[L,A]:
  def ! : L
  def ::(elem:A)(l: =>L): L
  def head(l: L): Option[A]
  def tail(l: L): L
  def ++(l: L)(l2: L): L

trait Treelike[T,A]:
  def ! : T
  def has(a:A)(lt: T, rt: T): T
  def root(t: T) : Option[A]
  def left(t: T) : T
  def right(t: T) : T
```

Programs for Testing: use All

```
def testList[L](implicit LL: Listlike[L,Int], ITRA: Iterable[L,Int]) = {
  val l = (3 :: !()) ++ (1 :: 2 :: !())
  println(sumElements(l))
  printElements(l)
}

def testTree[T](implicit TL: Treelike[T,Int], ITRA: Iterable[T,Int]) = {
  val t = 3.has(4.has(!(), !()), 2.has(!(),!()))
  println(sumElements(t))
  printElements(t)
}
```

Programs for Testing: use All

```
def testList[L](implicit LL: Listlike[L,Int], ITRA: Iterable[L,Int]) = {
  val l = LL.++(LL.::(3)(LL.!))(LL.::(1)(LL.::(2)(LL.!)))
  println(sumElements(l))
  printElements(l)
}

def testTree[T](implicit TL: Treelike[T,Int], ITRA: Iterable[T,Int]) = {
  val t = TL.has(3)(TL.has(4)(TL.!, TL.!), TL.has(2)(TL.!,TL.!))
  println(sumElements(t))
  printElements(t)
}
```

Implement Iter and Listlike for List

```
// behaves like Listlike[A] <: Iter[A] in OOP
given listIter[L,A](using LL: Listlike[L,A]): Iter[L,A] with
  extension (l: L)
    def getValue = l.head
    def getNext = l.tail

// behaves like List[A] <: Listlike[A] in OOP
given listListlike[A]: Listlike[List[A],A] with
  extension (u: Unit)
    def unary_! = Nil
  extension (a: A)
    def ::(l: =>List[A]) = a::l
  extension (l: List[A])
    def head = l.headOption
    def tail = l.tail
    def ++(l2: List[A]) = l ::: l2
```

Implement Iter and Listlike for List

```
// behaves like Listlike[A] <: Iter[A] in OOP
implicit def listIter[L,A](implicit LL: Listlike[L,A]): Iter[L,A] = new {
  def getValue(l: L) = LL.head(l)
  def getNext(l: L) = LL.tail(l)
}

// behaves like List[A] <: Listlike[A] in OOP
implicit def listListlike[A]: Listlike[List[A],A] = new {
  def ! = Nil
  def ::(a: A)(l: => List[A]) = a :: l
  def head(l: List[A]) = l.headOption
  def tail(l: List[A]) = l.tail
  def ++(l: List[A])(l2: List[A]) = l ::: l2
}
```

Implement Iterable for MyTree using Listlike,Iter

```
enum MyTree[+A]:
  case Leaf
  case Node(value: A, left: MyTree[A], right: MyTree[A])
import MyTree._

// behaves like MyTree[A] <: Iterable[A], but clumsy in OOP
given treeIterable[L,A](using LL: Listlike[L,A], _ITR: Iter[L,A])
  : Iterable[MyTree[A], A] with
  type Itr = L
  def ITR = _ITR
  extension (t: MyTree[A])
    def iter: L = t match {
      case Leaf => !()
      case Node(v, lt, rt) => v :: (lt.iter ++ rt.iter)
    }
```

Implement Iterable for MyTree using Listlike,Iter

```
enum MyTree[+A]:
  case Leaf
  case Node(value: A, left: MyTree[A], right: MyTree[A])
import MyTree._

// behaves like MyTree[A] <: Iterable[A], but clumsy in OOP
implicit def treeIterable[L,A](implicit LL: Listlike[L,A], _ITR: Iter[L,A])
  : Iterable[MyTree[A], A] = new {
  type Itr = L
  def ITR = _ITR
  def iter(t: MyTree[A]): L = t match {
    case Leaf => LL.!
    case Node(v, lt, rt) => LL.::(v)(LL.++(iter(lt))(iter(rt)))
  }
}
```

Implement Treelike for MyTree

```
// behaves like MyTree[A] <: Treelike[A] in OOP
given mytreeTreelike[A] : Treelike[MyTree[A],A] with
  extension (u: Unit)
    def unary_! = Leaf
  extension (a: A)
    def has(l: MyTree[A], r: MyTree[A]) = Node(a,l,r)
  extension (t: MyTree[A])
    def root = t match {
      case Leaf => None
      case Node(v,_,_) => Some(v)
    }
    def left = t match {
      case Leaf => t
      case Node(_,lt,_) => lt
    }
    def right = t match {
      case Leaf => t
      case Node(_,_,rt) => rt }
```

Implement Treelike for MyTree

```
// behaves like MyTree[A] <: Treelike[A] in OOP
implicit def mytreeTreelike[A] : Treelike[MyTree[A],A] = new {
  def ! = Leaf
  def has(a: A)(l: MyTree[A], r: MyTree[A]) = Node(a, l, r)
  def root(t: MyTree[A]) = t match {
    case Leaf => None
    case Node(v, _, _) => Some(v)
  }
  def left(t: MyTree[A]) = t match {
    case Leaf => t
    case Node(_, lt, _) => lt
  }
  def right(t: MyTree[A]) = t match {
    case Leaf => t
    case Node(_, _, rt) => rt
  }
}
```

Linking Modules

```
testList[List[Int]]

testTree[MyTree[Int]]
```

Test for Lazy List

```
def time[R](block: => R): R = {
  val t0 = System.nanoTime()
  val result = block    // call-by-name
  val t1 = System.nanoTime()
  println("Elapsed time: " + ((t1 - t0)/1000000) + "ms"); result
}
def sumN[I](n: Int, t: I)(implicit ITRA: Iterable[I,Int]): Int = {
  def go(res: Int, n: Int, itr: ITRA.Itr): Int =
    if (n <= 0) res
    else itr.getValue match {
      case None => res
      case Some(v) => go(v + res, n - 1, itr.getNext)
    }
  go(0, n, t.iter)
}
```

Test for Lazy List

```
def time[R](block: => R): R = {
  val t0 = System.nanoTime()
  val result = block    // call-by-name
  val t1 = System.nanoTime()
  println("Elapsed time: " + ((t1 - t0)/1000000) + "ms"); result
}
def sumN[I](n: Int, t: I)(implicit ITRA: Iterable[I,Int]): Int = {
  def go(res: Int, n: Int, itr: ITRA.Itr): Int =
    if (n <= 0) res
    else ITRA.ITR.getValue(itr) match {
      case None => res
      case Some(v) => go(v + res, n - 1, ITRA.ITR.getNext(itr))
    }
  go(0, n, ITRA.iter(t))
}
```

Test for Lazy List

```
def testTree2[T](implicit TL: Treelike[T,Int], ITRA: Iterable[T,Int]) = {
  def generateTree(n: Int): T = {
    def gen(lo: Int, hi: Int): T = {
      if (lo > hi) !()
      else {
        val mid = (lo + hi) / 2
        mid.has(gen(lo, mid - 1), gen(mid + 1, hi))
      }
    }
    gen(1, n)
  }

  // Problem: takes a few seconds to get a single value
  { val t = generateTree(200000)
    time (sumN(2, t)) }
}
```

Test for Lazy List

```
def testTree2[T](implicit TL: Treelike[T,Int], ITRA: Iterable[T,Int]) = {
  def generateTree(n: Int): T = {
    def gen(lo: Int, hi: Int): T = {
      if (lo > hi) TL.!
      else {
        val mid = (lo + hi) / 2
        TL.has(mid)(gen(lo, mid - 1), gen(mid + 1, hi))
      }
    }
    gen(1, n)
  }

  // Problem: takes a few seconds to get a single value
  { val t = generateTree(200000)
    time (sumN(2, t)) }
}
```

Lazy List

```
sealed abstract class LazyList[+A] {
  def matches[R](caseNil: =>R, caseCons: (A,LazyList[A])=>R) : R
}
case object LNil extends LazyList[Nothing] {
  def matches[R](caseNil: =>R, _u: (Nothing,LazyList[Nothing])=>R) =
    caseNil
}
class LCons[A](hd: A, _tl: =>LazyList[A]) extends LazyList[A] {
  lazy val tl = _tl
  def matches[R](_u: =>R, caseCons: (A, LazyList[A])=>R) =
    caseCons(hd, tl)
}
object LazyList {
  extension [A](l: LazyList[A])
    def append(l2: LazyList[A]) : LazyList[A] =
      l.matches(l2, (hd,tl) => LCons(hd, tl.append(l2)))
}
import LazyList.*
```

Lazy List

```
sealed abstract class LazyList[+A] {
  def matches[R](caseNil: =>R, caseCons: (A,LazyList[A])=>R) : R
}
case object LNil extends LazyList[Nothing] {
  def matches[R](caseNil: =>R, _u: (Nothing,LazyList[Nothing])=>R) =
    caseNil
}
class LCons[A](hd: A, _tl: =>LazyList[A]) extends LazyList[A] {
  lazy val tl = _tl
  def matches[R](_u: =>R, caseCons: (A, LazyList[A])=>R) =
    caseCons(hd, tl)
}
object LazyList {
  def append[A](l: LazyList[A])(l2: LazyList[A]) : LazyList[A] =
    l.matches(l2, (hd,tl) => LCons(hd, append(tl)(l2)))
}
import LazyList.*
```

Lazy List

```
given lazylistListlike[A]: Listlike[LazyList[A],A] with
  extension (u: Unit)
    def unary_! = LNil
  extension (a: A)
    def ::(l: =>LazyList[A]) = LCons(a,l)
  extension (l: LazyList[A])
    def head = l.matches(None, (hd,tl) => Some(hd))
    def tail = l.matches(LNil, (hd,tl)=>tl)
    def ++(l2: LazyList[A]) = l.append(l2)


testList[LazyList[Int]]
testTree[MyTree[Int]]
testTree2[MyTree[Int]]
```

Lazy List

```
implicit def lazylistListlike[A]: Listlike[LazyList[A],A] = new {
  def ! = LNil
  def ::(a: A)(l: => LazyList[A]) = LCons(a, l)
  def head(l: LazyList[A]) = l.matches(None, (hd, tl) => Some(hd))
  def tail(l: LazyList[A]) = l.matches(LNil, (hd, tl) => tl)
  def ++(l: LazyList[A])(l2: LazyList[A]) = LazyList.append(l)(l2)
}

testList[LazyList[Int]]
testTree[MyTree[Int]]
testTree2[MyTree[Int]]
```

Type class: Code Reuse

IntStack Spec

```
trait Stack[S,A]:
  extension (u: Unit)
    def empty : S
  extension (s: S)
    def get: (A,S)
    def put(a: A): S

def testStack[S](implicit STK: Stack[S,Int]) = {
  val s = ().empty.put(3).put(-2).put(4)
  val (v1,s1) = s.get
  val (v2,s2) = s1.get
  (v1,v2)
}
```

IntStack Spec

```
trait Stack[S,A]:
  def empty : S
  def get(s: S): (A,S)
  def put(s: S)(a: A): S

def testStack[S](implicit STK: Stack[S,Int]) = {
  val s = STK.put(STK.put(STK.put(STK.empty)(3))(-2))(4)
  val (v1,s1) = STK.get(s)
  val (v2,s2) = STK.get(s1)
  (v1,v2)
}
```

Implementation using List

```
given BasicStack[A] : Stack[List[A],A] with
  extension (u: Unit)
    def empty = List()
  extension (s: List[A])
    def get = (s.head, s.tail)
    def put(a: A) = a :: s
```

Implementation using List

```
implicit def BasicStack[A] : Stack[List[A],A] = new {
  def empty = List()
  def get(s: List[A]) = (s.head, s.tail)
  def put(s: List[A])(a: A) = a :: s
}
```

Modifying Traits

```
def StackOverridePut[S,A](newPut: (S,A)=>S)(implicit STK: Stack[S,A])
: Stack[S,A] = new {
  extension (u: Unit)
    def empty = STK.empty(u)
  extension (s: S)
    def get = STK.get(s)
    def put(a: A) = newPut(s,a)
}

def Doubling[S](implicit STK: Stack[S,Int]) : Stack[S,Int] =
  StackOverridePut((s,a) => s.put(2 * a))

def Incrementing[S](implicit STK: Stack[S,Int]) : Stack[S,Int] =
  StackOverridePut((s,a) => s.put(a + 1))

def Filtering[S](implicit STK: Stack[S,Int]) : Stack[S,Int] =
  StackOverridePut((s,a) => if (a >= 0) s.put(a) else s)
```

Modifying Traits

```
def StackOverridePut[S,A](newPut: (S,A)=>S)(implicit STK: Stack[S,A])
: Stack[S,A] = new {
  def empty = STK.empty
  def get(s: S) = STK.get(s)
  def put(s: S)(a: A) = newPut(s,a)
}

def Doubling[S](implicit STK: Stack[S,Int]) : Stack[S,Int] =
  StackOverridePut((s,a) => STK.put(s)(2 * a))

def Incrementing[S](implicit STK: Stack[S,Int]) : Stack[S,Int] =
  StackOverridePut((s,a) => STK.put(s)(a + 1))

def Filtering[S](implicit STK: Stack[S,Int]) : Stack[S,Int] =
  StackOverridePut((s,a) => if (a >= 0) STK.put(s)(a) else s)
```

Linking

```
// testStack(BasicStack)
testStack

// testStack(Filtering(Incrementing (Doubling(BasicStack))))
testStack(Filtering (Incrementing (Doubling)))

// testStack(Filtering(Incrementing(Incrementing(Doubling(BasicStack)))))
testStack(Filtering (Incrementing (Incrementing (Doubling))))
```

Implementation: Sorted Stack

```
def SortedStack : Stack[List[Int],Int] = new {
  extension (u: Unit)
    def empty = List()
  extension (s: List[Int])
    def get = (s.head, s.tail)
    def put(a: Int) : List[Int] = {
      def loop(l: List[Int]) : List[Int] = l match {
        case Nil => a :: Nil
        case hd :: tl => if (a <= hd) a :: l else hd :: loop(tl)
      }
      loop(s)
    }
  }

testStack(Filtering(Incrementing(Doubling(SortedStack))))
```

Implementation: Sorted Stack

```
def SortedStack : Stack[List[Int],Int] = new {
  def empty = List()
  def get(s: List[Int]) = (s.head, s.tail)
  def put(s: List[Int])(a: Int) : List[Int] = {
    def loop(l: List[Int]) : List[Int] = l match {
      case Nil => a :: Nil
      case hd :: tl => if (a <= hd) a :: l else hd :: loop(tl)
    }
    loop(s)
  }
}

testStack(Filtering(Incrementing(Doubling(SortedStack))))
```

Higher Type Classes

Interfaces I

```
// eg. Iter[List]
trait Iter[I[_]]:
  extension [A](i: I[A])
    def getValue: Option[A]
    def getNext: I[A]

// eg. Iterable[MyTree]
trait Iterable[I[_]]:
  type Itr[_]
  given ITR: Iter[Itr]
  extension [A](i: I[A])
    def iter: Itr[A]

given iter2iterable[I[_]](using _ITR: Iter[I]): Iterable[I] with
  type Itr[A] = I[A]
  def ITR = _ITR
  extension [A](i:I[A])
    def iter = i
```

```
// trait Iter[I,A]:
//   extension (i: I)
//     def getValue: Option[A]
//     def getNext: I

// trait Iterable[I,A]:
//   type Itr
//   given ItrI: Iter[Itr,A]
//   extension (i: I)
//     def iter: Itr
```

Interfaces I

```
// eg. Iter[List]
trait Iter[I[_]]:
  def getValue[A](i: I[A]): Option[A]
  def getNext[A](i: I[A]): I[A]

// eg. Iterable[MyTree]
trait Iterable[I[_]]:
  type Itr[_]
  implicit def ITR: Iter[Itr]
  def iter[A](i: I[A]): Itr[A]

implicit def iter2iterable[I[_]](using _ITR: Iter[I]): Iterable[I] = new {
  type Itr[A] = I[A]
  def ITR = _ITR
  def iter[A](i:I[A]) = i
}
```

Programs for Testing: use Iter, Iterable

```
def sumElements[I[_]](xs: I[Int])(implicit ITRA:Iterable[I]) = {
  def loop(i: ITRA.Itr[Int]): Int =
    i.getValue match {
      case None => 0
      case Some(n) => n + loop(i.getNext)
    }
  loop(xs.iter)
}

def printElements[I[_],A](xs: I[A])(implicit ITRA: Iterable[I]) = {
  def loop(i: ITRA.Itr[A]): Unit =
    i.getValue match {
      case None =>
      case Some(a) => {println(a); loop(i.getNext)}
    }
  loop(xs.iter)
}
```

Programs for Testing: use Iter, Iterable

```
def sumElements[I[_]](xs: I[Int])(implicit ITRA:Iterable[I]) = {
  def loop(i: ITRA.Itr[Int]): Int =
    ITRA.ITR.getValue(i) match {
      case None => 0
      case Some(n) => n + loop(ITRA.ITR.getNext(i))
    }
  loop(ITRA.iter(xs))
}

def printElements[I[_],A](xs: I[A])(implicit ITRA: Iterable[I]) = {
  def loop(i: ITRA.Itr[A]): Unit =
    ITRA.ITR.getValue(i) match {
      case None =>
      case Some(a) => {println(a); loop(ITRA.ITR.getNext(i))}
    }
  loop(ITRA.iter(xs))
}
```

Interfaces II

```
trait Listlike[L[_]]:
  extension[A](u:Unit)
    def unary_! : L[A]
  extension[A](elem:A)
    def ::(l: =>L[A]): L[A]
  extension[A](l: L[A])
    def head: Option[A]
    def tail: L[A]
    def ++(l2: L[A]): L[A]

trait Treelike[T[_]]:
  extension[A](u:Unit)
    def unary_! : T[A]
  extension[A](a:A)
    def has(lt: T[A], rt: T[A]): T[A]
  extension[A](t: T[A])
    def root : Option[A]
    def left : T[A]
    def right : T[A]
```

Interfaces II

```
trait Listlike[L[_]]:
  def ![A] : L[A]
  def ::[A](elem:A)(l: =>L[A]): L[A]
  def head[A](l: L[A]): Option[A]
  def tail[A](l: L[A]): L[A]
  def ++[A](l: L[A])(l2: L[A]): L[A]

trait Treelike[T[_]]:
  def ![A] : T[A]
  def has[A](a:A)(lt: T[A], rt: T[A]): T[A]
  def root[A](t: T[A]) : Option[A]
  def left[A](t: T[A]) : T[A]
  def right[A](t: T[A]) : T[A]
```

Programs for Testing: use All

```
def testList[L[_]](implicit LL: Listlike[L], ITRA: Iterable[L]) = {
  val l = (3 :: !()) ++ (1 :: 2 :: !())
  println(sumElements(l))
  printElements(l)
}

def testTree[T[_]](implicit TL: Treelike[T], ITRA: Iterable[T]) = {
  val t = 3.has(4.has(!(), !()), 2.has(!(),!()))
  println(sumElements(t))
  printElements(t)
}
```

Programs for Testing: use All

```
def testList[L[_]](implicit LL: Listlike[L], ITRA: Iterable[L]) = {
  val l = LL.++(LL.::(3)(LL.!))(LL.::(1)(LL.::(2)(LL.!)))
  println(sumElements(l))
  printElements(l)
}

def testTree[T[_]](implicit TL: Treelike[T], ITRA: Iterable[T]) = {
  val t = TL.has(3)(TL.has(4)(TL.!, TL.!), TL.has(2)(TL.!,TL.!))
  println(sumElements(t))
  printElements(t)
}
```

List: provide Iter, ListIF

```
// behaves like List[A] <: Iter[A] in OOP
given listIter: Iter[List] with
  extension [A](l: List[A])
    def getValue = l.headOption
    def getNext = l.tail

// behaves like List[A] <: Listlike[A] in OOP
given listListlike: Listlike[List] with
  extension [A](u: Unit)
    def unary_! = Nil
  extension [A](a: A)
    def ::(l: =>List[A]) = a::l
  extension [A](l: List[A])
    def head = l.headOption
    def tail = l.tail
    def ++(l2: List[A]) = l ::: l2
```

List: provide Iter, ListIF

```
// behaves like List[A] <: Iter[A] in OOP
implicit def listIter: Iter[List] = new {
  def getValue[A] (l: List[A]) = l.headOption
  def getNext[A] (l: List[A]) = l.tail
}
// behaves like List[A] <: Listlike[A] in OOP
implicit def listListlike: Listlike[List] = new {
  def ![A] = Nil
  def ::[A](a: A)(l: => List[A]) = a :: l
  def head[A](l: List[A]) = l.headOption
  def tail[A](l: List[A]) = l.tail
  def ++[A](l: List[A])(l2: List[A]) = l ::: l2
}
```

MyTree: use Iter, ListIF, provide Iterable, TreeIF

```
enum MyTree[+A]:
  case Leaf
  case Node(value: A, left: MyTree[A], right: MyTree[A])
import MyTree._

// behaves like MyTree[A] <: Iterable[A], but clumsy in OOP
given treeIterable[L[_]](using LL: Listlike[L], _ITR: Iter[L]): Iterable[MyTree]
with
  type Itr[A] = L[A]
  def ITR = _ITR
  extension [A](t: MyTree[A])
    def iter: L[A] = t match {
      case Leaf => !()
      case Node(v, lt, rt) => v :: (lt.iter ++ rt.iter)
    }
```

MyTree: use Iter, ListIF, provide Iterable, TreeIF

```
enum MyTree[+A]:
  case Leaf
  case Node(value: A, left: MyTree[A], right: MyTree[A])
import MyTree._

// behaves like MyTree[A] <: Iterable[A], but clumsy in OOP
implicit def treeIterable[L[_]](using LL: Listlike[L], _ITR: Iter[L]):
Iterable[MyTree] = new {
  type Itr[A] = L[A]
  def ITR = _ITR
  def iter[A] (t: MyTree[A]): L[A] = t match {
    case Leaf => LL.!
    case Node(v, lt, rt) => LL.::(v)(LL.++(iter(lt))(iter(rt)))
  }
}
```

MyTree: use Iter, ListIF, provide Iterable, TreeIF

```
// behaves like MyTree[A] <: Treelike[A] in OOP
given mytreeTreelike: Treelike[MyTree] with
  extension [A](u: Unit)
    def unary_! = Leaf
  extension [A](a: A)
    def has(l: MyTree[A], r: MyTree[A]) = Node(a,l,r)
  extension [A](t: MyTree[A])
    def root = t match {
      case Leaf => None
      case Node(v,_,_) => Some(v)
    }
    def left = t match {
      case Leaf => t
      case Node(_,lt,_) => lt
    }
    def right = t match {
      case Leaf => t
      case Node(_,_,rt) => rt }
```

MyTree: use Iter, ListIF, provide Iterable, TreeIF

```
// behaves like MyTree[A] <: Treelike[A] in OOP
implicit def mytreeTreelike: Treelike[MyTree] = new {
  def ![A] = Leaf
  def has[A] (a: A)(l: MyTree[A], r: MyTree[A]) = Node(a, l, r)
  def root[A](t: MyTree[A]) = t match {
    case Leaf => None
    case Node(v, _, _) => Some(v)
  }
  def left[A](t: MyTree[A]) = t match {
    case Leaf => t
    case Node(_, lt, _) => lt
  }
  def right[A](t: MyTree[A]) = t match {
    case Leaf => t
    case Node(_, _, rt) => rt
  }
}
```

Linking Modules

```
testList[List]

testTree[MyTree]
```

List with Map

```
trait Maplike[L[_]]:
  extension[A](l: L[A])
    def map[B](f: A => B): L[B]

def testMapList[L[_]](implicit LL: Listlike[L], ML: Maplike[L], ITR: Iter[L]) = {
  val l1 = 3.3 :: 2.2 :: 1.5 :: !()
  val l2 = l1.map((n:Double)=>n.toInt)
  val l3 = l2.map((n:Int)=>n.toString)
  printElements(l3)
}
```

List with Map

```
trait Maplike[L[_]]:
  def map[A,B](l: L[A])(f: A => B): L[B]

def testMapList[L[_]] (implicit LL: Listlike[L], ML: Maplike[L], ITR: Iter[L]) = {
  val l1 = LL.::(3.3)(LL.::(2.2)(LL.::(1.5)(LL.!)))
  val l2 = ML.map(l1)((n:Double)=>n.toInt)
  val l3 = ML.map(l2)((n:Int)=>n.toString)
  printElements(l3)
}
```

List with Map

```
given listMaplike: Maplike[List] with
extension [A](l: List[A])
  def map[B](f: A => B) = l.map(f)

testMapList[List]
```

List with Map

```
implicit def listMaplike: Maplike[List] = new {
  def map[A,B](l: List[A])(f: A => B) = l.map(f)
}

testMapList[List]
```

Turning Type Classes into OO Classes

Interfaces

```
trait DataProcessor[D]:
  extension (d: D)
    def input(s: String) : D
    def output : String

trait DPFactory:
  extension (u: Unit)
    def getTypes: List[String]
    def makeDP(dptype: String) : ???

def run(implicit factory: DPFactory) : Unit
```

How to return data with associated functions like OOP?

Turning Type Classes into OO Classes

```
import scala.language.implicitConversions
type curry1[F[_,_],A1] = ([X] =>> F[X,A1])
type curry2[F[_,_,_],A1,A2] = ([X] =>> F[X,A1,A2])
type curry3[F[_,_,_,_],A1,A2,A3] = ([X] =>> F[X,A1,A2,A3])

trait dyn[S[_]]:
  type Data
  val * : Data
  given DI: S[Data]

object dyn {
  implicit // needed for implicit conversion of D into dyn[S]
  def apply[S[_],D](d: D)(implicit i: S[D]): dyn[S] = new {
      type Data = D
      val * = d
      val DI = i
    }
}
```

Turning Type Classes into OO Classes

```scala
import scala.language.implicitConversions
type curry1[F[_,_],A1] = ([X] =>> F[X,A1])
type curry2[F[_,_,_],A1,A2] = ([X] =>> F[X,A1,A2])
type curry3[F[_,_,_,_],A1,A2,A3] = ([X] =>> F[X,A1,A2,A3])

trait dyn[S[_]]:
  type Data
  val * : Data
  implicit def DI: S[Data]

object dyn {
  implicit // needed for implicit conversion of D into dyn[S]
  def apply[S[_],D](d: D)(implicit i: S[D]): dyn[S] = new {
    type Data = D
    val * = d
    val DI = i
  }
}
```

Interfaces

```
trait DataProcessor[D]:
  extension (d: D)
    def input(s: String): D
    def output: String

trait DPFactory:
  extension (u: Unit)
    def getTypes: List[String]
    def makeDP(dptype: String): dyn[DataProcessor]
```

Interfaces

```
trait DataProcessor[D]:
  def input(d: D)(s: String): D
  def output(d: D): String

trait DPFactory:
  def getTypes: List[String]
  def makeDP(dptype: String): dyn[DataProcessor]
```

Test

```
def test(implicit DF: DPFactory) = {
  def go(types: List[String]) : Unit =
    types match {
      case Nil => ()
      case ty :: rest => {
        val dp = ().makeDP(ty)
        println(dp.*.input("10").input("20").output)
        go(rest)
      }
    }
  val types = ().getTypes
  println(types)
  go(types)
}
```

Test

```
def test(implicit DF: DPFactory) = {
  def go(types: List[String]) : Unit =
    types match {
      case Nil => ()
      case ty :: rest => {
        val dp : dyn = DF.makeDP(ty)
        println(dp.DI.output(dp.DI.input(dp.DI.input(dp.*)("10"))("20")))
        go(rest)
      }
    }
  val types = DF.getTypes
  println(types)
  go(types)
}
```

Data Processor

```
given dpfactory: DPFactory with
  extension (u: Unit)
    def getTypes = List("sum", "mult")
    def makeDP(dptype: String) = {
      if (dptype == "sum")
        makeProc(0, (x, y) => x + y)
      else
        makeProc(1, (x, y) => x * y)
    }

    def makeProc(init: Int, op: (Int, Int) => Int): dyn[DataProcessor] = {
      given dp: DataProcessor[Int] with
        extension (d: Int)
          def input(s: String) = op(d, s.toInt)
          def output = d.toString()
      init       // dyn(init) // dyn.apply[Int,DataProcessor](init)(dp)
    }
```

Data Processor

```
implicit val dpfactory: DPFactory = new {
  def getTypes = List("sum", "mult")
  def makeDP(dptype: String) = {
    if (dptype == "sum")
      makeProc(0, (x, y) => x + y)
    else
      makeProc(1, (x, y) => x * y)
  }

  def makeProc(init: Int, op: (Int, Int) => Int): dyn[DataProcessor] = {
    implicit def dp: DataProcessor[Int] = new {
      def input(d: Int)(s: String) = op(d, s.toInt)
      def output(d: Int) = d.toString()
    }
    init // dyn(init)(dp) // dyn.apply[Int,DataProcessor](init)(dp)
  }
}
```

Heterogeneous List of Iter

```
trait Iter[I,A]:
  extension (i: I)
    def getValue: Option[A]
    def getNext: I

def sumElements[I](xs: I)(implicit ITR:Iter[I,Int]) : Int = {
  xs.getValue match {
    case None => 0
    case Some(n) => n + sumElements(xs.getNext)
  }
}

def sumElementsList(xs: List[dyn[curry1[Iter,Int]]]) : Int =
  xs match {
    case Nil => 0
    case hd :: tl => sumElements(hd.*) + sumElementsList(tl)
  }
```

Heterogeneous List of Iter

```
trait Iter[I,A]:
  def getValue(i: I): Option[A]
  def getNext(i: I): I

def sumElements[I](xs: I)(implicit ITR:Iter[I,Int]) : Int = {
  ITR.getValue(xs) match {
    case None => 0
    case Some(n) => n + sumElements(ITR.getNext(xs))
  }
}

def sumElementsList(xs: List[dyn[curry1[Iter,Int]]]) : Int =
  xs match {
    case Nil => 0
    case hd :: tl => sumElements(hd.*) + sumElementsList(tl)
  }
```

Test

```
given listIter[A]: Iter[List[A],A] with
  extension (l: List[A])
    def getValue = l.headOption
    def getNext = l.tail

given decIter : Iter[Int,Int] with
  extension (i: Int)
    def getValue = if (i >= 0) Some(i) else None
    def getNext = i - 1

sumElementsList(List(
  100,
  List(1,2,3),
  10))
```

Test

```
implicit def listIter[A]: Iter[List[A],A] = new {
  def getValue(l: List[A]) = l.headOption
  def getNext(l: List[A]) = l.tail
}

implicit val decIter : Iter[Int,Int] = new {
  def getValue(i: Int) = if (i >= 0) Some(i) else None
  def getNext(i: Int) = i - 1
}

sumElementsList(List(
  100,
  List(1,2,3),
  10))
```

PART 4

Imperative Programming with Memory Updates

Mutable Update in Scala

Mutable Variables

- Mutable Variables
  - Use “var” instead of “val” and “def”
  - We can update the value stored in a variable.

```
class Main(i: Int) {
  var a = i
}

val m = new Main(10)
m.a  // 10
m.a = 20
m.a  // 20
m.a += 5  // m.a = m.a + 5
m.a  // 25
```

While loop

- While loop
  - Syntax: `while (`*cond*`)` *body*
    Executes *body* while *cond* holds.
  - It is equivalent to:

```
def mywhile(cond: =>Boolean)(body: =>Unit) : Unit =
  if (cond) { body; mywhile(cond)(body) } else ()
```

- Example

```
var i = 0
var sum = 0
while (i <= 100) { // mywhile (i <= 100) {
  sum += i
  i += 2
}
sum // 2550
```

For loop

- For loop
  - Syntax: `for (i <- ` *collection* `)` *body*
    Executes *body* for each `i` in *collection*.
  - It is equivalent to:

```
def myfor[A](xs: Traversable[A])(f: A => Unit) : Unit =
  xs.foreach(f)
```

- Example

```
var sum = 0
for (i <- 0 to 100 by 2) { // myfor (0 to 100 by 2) { i =>
  sum += i
}
sum // 2550
```

Immutability, Mutability & Ownership

Immutability for Guarantee & Mutability for Efficiency

- Immutable array

  arr = Array.new([1;2;3])

  … using arr …

  arr’ = Array.set(arr, 0, 42)

  f(arr’)

  … using arr’ …

  g()

  … using arr’ …

- Mutable array

  arr = Array.new([1;2;3])

  … using arr …

  Array.set(arr, 0, 42);

  f(arr)

  … using arr …

  g()

  … using arr …

Can we not have both? Mutability with Ownership!

- Immutable array

```
arr = Array.new([1;2;3])

… using arr …

arr' = Array.set(arr, 0, 42)

f(arr')

… using arr' …

g()

… using arr' …
```

- Mutation with Ownership

```
// own
arr = Array.new([1;2;3])

… using arr …
// mutable borrow
Array.set(&mut arr, 0, 42)
// immutable borrow
f(&arr)

… using arr …

g()

… using arr …
```

Types with Ownership

- Ownership Types

  - expresses and guarantees immutability
    - making code behavior more predictable

  - automatically deallocates memory when its ownership has gone
    - guaranteeing absence of use-after-free, double-free, memory-leak

  - disallows mutating the same value at the same time
    - guaranteeing data-race freedom in concurrent code

Programming with Mutation in a Principled Way!

Types, Values & Memory

Values and Types

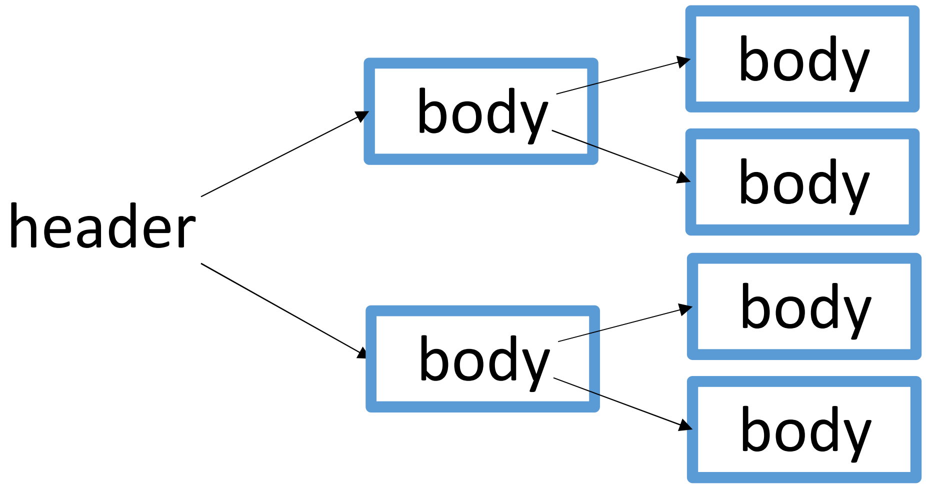
A type defines a set of values.

- For functionality, we only need to know the denotation of values.
- For efficiency, we also need to know the shape of values.

A value has a tree structure and consists of a header and bodies.

- A header is data that may contain scalar values and pointers to bodies.
- A body is data that must be stored in memory and may contain scalar values and pointers to other bodies.
- When passing or storing a value, only its header is passed or stored.

Types in Rust: i64, u64, (T1,T2), [T; 5], String, Vec<T>, …



Stack memory with static size and static lifetime

Stack memory consists of variables of sized types (ie, static header sizes) with scopes (ie, static lifetime).

A stack variable of a type T stores a header of the type T.

```
// variable arg of type i64 (8-byte header) with Scope 1
fn foo(arg: i64) -> i64 {
    let mut x = 10;   // variable x of type i64 with Scope 1
    {
        let y = arg + x;   // variable y of type i64 with Scope 2
        x = x + y;
    }  // Scope 2
    let x2 = x; // variable x2 of type i64 with Scope 1
    let z = arg + x2;   // variable z of type i64 with Scope 1
    return z;
}  // Scope 1
```

Heap memory with dynamic size and dynamic lifetime

Heap memory consists of blocks of all types (ie, dynamic sizes) with no scopes (ie, dynamic lifetime).
A heap block can be a body of a value, which can also store headers of other values.

```
// variable sz of type usize (8-byte header) with Scope 1
fn gee(sz: usize) -> Vec<i64> {
// variable v of type Vec<i64> (24-byte header) with Scope 1
// The header points to a heap block of (sz * 8) bytes, filled with zeros
    let v : Vec<i64> = vec![0; sz];
// The header goes out of Scope 1 and the heap block is still alive
    return v;
}  // Scope 1
```

Principles of Ownership & Lifetime

Issues with mutation and deallocation for memory

<Problematic mutations>
Multiple parties modify a value logically simultaneously.

- Problem
  The value may become inconsistent despite valid individual updates.

A value is read logically while being modified.

- Problem
  An inconsistent intermediate state may be read.

<Problematic deallocation>
A value's body is deallocated while its header remains accessible.

- Problem
  The deallocated body may be accessed via the header (use-after-free bug).

Principles of Ownership

<Ownership rules>

- A header must be stored in exactly one location, which owns the header.
- Every heap block must be owned by exactly one header that directly points to it.
- Headers can be moved between locations.
- Headers are dropped when their owner is freed or overwritten.
- When dropped, a header's owned heap blocks are also freed.
- A stack variable is freed when it goes out of scope.

<Borrowing rules> (known as, “mutation-xor-sharing” checking)

- Mutable: single borrower has exclusive read/write access.
- Immutable: multiple borrowers and owner can read.

Principles of Lifetime

<Lifetime definition>

- Stack variables have a static lifetime bound by their scope.
- Heap blocks have a dynamic lifetime that ends when their owner is dropped.
- Headers have a dynamic lifetime that ends when their owner is freed or overwritten.

<Lifetime rules for borrowing> (known as, "borrow lifetime checking")

- A location's lifetime must be longer than any of its borrowers' lifetimes.

An example with ownership and borrowing

An example showing how ownership and borrowing work.

```
// arg owns a string, say ARG
fn gee(arg: String) -> String {
    let mut x = String::from("abc"); // x owns the string "abc"
    {
        let y = x + &arg;  // y owns the string "abc"+ARG, x not accessible
        x = y + &arg;  // x owns the string "abc"+ARG+ARG, y not accessible
    } // y is deallocated
    let z = x; // z owns the string "abc"+ARG+ARG, x not accessible
    return z; // the string "abc"+ARG+ARG is returned, z not accessible
} // arg, x, z are dallocated, the string ARG in arg is deallocated
```

Ownership and Lifetime in Rust

Rust's approach

- References to a location of type T:
  - &T: immutable reference
  - &mut T: mutable reference

- Function signatures specify ownership/lifetime conditions

- Compiler checks ownership/lifetime in Safe Rust at two levels:
  - Stack variables (assuming function signatures)
  - Function implementations (verifying against signatures)

- Heap Memory Management:
  - Experts implement primitive heap types in Unsafe Rust with ownership/lifetime guarantees specified in function signatures
  - Users compose these primitives in Safe Rust to build complex data structures

Datatypes with Ownership & Lifetime

Scalar Types

```
fn main() {
    let mut x = [1,2,3,4];
    println!("{}", x[2]);
    x[2] = 42;
    println!("{}", x[2]);
    let y = x;
    println!("{}", x[0]);
    println!("{}", y[0]);
}
```

String and str types

```
fn main() {
    let mut s : &str = "abc";
    {
        let mut st : String = String::from(s);
        st.push_str("def");   // String::push_str(&mut st, "def");
        println!("{}", st);
        let s2 : &str = &st[1..4];
        // st.push_str("ghi");
        println!("{}", s2);
        // s = s2;
    }
    // println!("{}", s);
}
```

Function Types

```
fn foo(s: &String) -> &str {
    return &s[0..2];
}
fn gee(f: fn(&String) -> &str) {
    let s;
    // {
        let a = String::from("test");
        s = f(&a);
    // }
    println!("{}", s);
}
fn main() {
    gee(foo);
}
```

Function Types with lifetime annotation

```
fn foo<'a>(s: &'a String) -> &'a str {
    return &s[0..2];
}
fn gee (f: for <'a> fn(&'a String) -> &'a str) {
    let s;
    // {
        let a = String::from("test");
        s = foo(&a);
    // }
    println!("{}", s);
}
fn main() {
    gee(foo);
}
```

General lifetime annotation

```
fn foo1<'a>(s1: &'a String, s2: &'a String) -> &'a str {
    …
}
fn foo2<'a,'b>(s1: &'a String, s2: &'b String) -> &'a str {
    …
}
fn foo3<'a,'b,'c>(s1: &'a String, s2: &'b String) -> &'c str {
    …
}
fn foo4<'a,'b,'c, 'd, 'e>(s1: &'a String, s2: &'b String, s3: &'c String)
     -> (&'d str, &'e str)
where 'a: 'd, 'b:'d, 'b:'e, 'c:'e {
    …
}
```

General lifetime annotation

```
fn foo1<'a>(s1: &'a String, s2: &'a String) -> &'a str {
    return if rand::random() { &s1[0..1] } else { &s2[0..2] }; }
fn foo2<'a,'b>(s1: &'a String, s2: &'b String) -> &'a str {
    return &s1[0..1]; }
fn foo3<'a,'b,'c>(s1: &'a String, s2: &'b String) -> &'c str {
    return "abc"; }
fn foo4<'a,'b,'c, 'd, 'e>(s1: &'a String, s2: &'b String, s3: &'c String)
     -> (&'d str, &'e str)
where 'a: 'd, 'b:'d, 'b:'e, 'c:'e {
    let r1 = if rand::random() { &s1[0..1] } else { &s2[0..1] };
    let r2 = if rand::random() { &s2[0..1] } else { &s3[0..1] };
    (r1,r2)
}
```

Struct

```
struct User {
    active: bool,
    name: String,
}
fn main() {
    println!("Size: {} bytes", std::mem::size_of::<User>());
    let mut user_name = String::from("gil");
    let mut u = User { active: true, name: user_name };
    u.name.push_str(" hur");
    println!("{}, {}", u.active, u.name);
    user_name = u.name;
    println!("{}, {}", u.active, user_name);
    // println!("{} {}", u.active, u.name);
}
```

Struct with lifetime annotation

```
// Key Idea: Track the lifetime of each field of a struct separately.
struct User<'a,'b> { name: &'a mut String, email: &'b String }
fn main() {
    println!("Size: {} bytes", std::mem::size_of::<User>());
    let mut user_name = String::from("gil");
    let temp : &String;
    {   let mut user_email = String::from("gil.hur@sf.snu.ac.kr");
        let mut u = User { name: &mut user_name, email: &user_email };
        u.name.push_str(" hur");
        println!("{}, {}", u.name, u.email);
        temp = u.name; // temp = u.email;
        // u.name.push_str("xxx");
    }
    println!("{}", temp);
}
```

Struct with lifetime annotation (Variations)

```
// Key Idea: Track the lifetime of each field of a struct separately.
// Easy to understand if you inline the definition of a struct.
struct User1<'a> {
  name: &'a String,
  email: &'a String }
// User1 : for <'a> fn(&'a String, &'a String) : User1<'a>
// User1::name : for <'a> fn(User1<'a>) : &'a String
// User1::email : for <'a> fn(User1<'a>) : &'a String

struct User2<'a,'b> {
  user: User1<'a>,
  addr: &'b String }
// User2 : for <'a,'b> fn(User1<'a>, &'b String) : User2<'a,'b>
// User2::user : for <'a,'b> fn(User2<'a,'b>) : User1<'a>
// User2::addr : for <'a,'b> fn(User2<'a,'b>) : &'b String
```

Struct with lifetime annotation (Variations)

```
struct Foo<'a,'b> {
    x: &'a i32,
    y: &'b i32,
}

fn main() {
    let x = 1;
    let v;
    {
        let y = 2;
        let f = Foo { x: &x, y: &y };
        v = f.x;
    }
    println!("{}", *v);
}
```

Enum

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}
use IpAddr::*;
fn main() {
    let ip1 = V4(147,36,52,255);
    match ip1 {  // match &ip1 // match &mut ip1
        V4(a1,a2,a3,a4) => println!("{a1:?} {a2:?} {a3:?} {a4:?}"),
        V6(addr) => println!("{addr:?}")   }
    let ip2 = V6(String::from("123.123.123.123.123.123"));
    match &ip2 {
        V4(a1,a2,a3,a4) => println!("{a1:?} {a2:?} {a3:?} {a4:?}"),
        V6(addr) => println!("{addr:?}")   }
}
```

Recursive Enum with Box

```
enum List<T> { Nil, Cons(T, Box<List<T>>) }
impl <T> List<T> {
    fn new() -> List<T> { List::Nil }
    fn cons(hd: T, tl: List<T>) -> List<T> {  List::Cons(hd, Box::new(tl)) }
}
fn mylen<T>(l: &List<T>) -> u64 {
    match l {
        List::Nil => 0,
        List::Cons(_, tl) => 1+mylen(tl) // 1+mylen(tl.as_ref())
    }
}
fn main() {
    let l = List::cons(0, List::cons(1, List::cons(2, List::new())));
    println!("{}", mylen(&l));
}
```

Update via a mutable reference

```
use std::mem;
fn test(x: &mut List<i32>) {
    // *x = List::cons(0, *x)
    let old = mem::replace(x, List::new());
    *x = List::cons(0, old)
}

fn main() {
    let mut l = List::cons(0, List::cons(1, List::cons(2, List::new())));
    println!("{}", mylen(&l));
    test(&mut l);
    println!("{}", mylen(&l));
}
```

Smart Pointers

Box type

```
enum List<T> {
  Nil,
  Cons(T, Box<List<T>>)
}
```

Rc type

Use Rc type to replace static borrow lifetime checking with dynamic checking.

<Problem>

```
enum List<T> {
    Cons(T, Rc<List<T>>),
    Nil,
}
use crate::List::*;
fn test() -> (List<i64>,List<i64>) {
    let a = Cons(5, Box::new(Nil));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
    (b,c)
}
```

Rc type

```
use std::rc::Rc;
enum List<T> { Cons(T, Rc<List<T>>), Nil, }
use crate::List::*;
impl <T> List<T> {
   fn unfold(&self) -> Option<(&T,&List<T>)> {
      if let Cons(hd,tl) = self { Some((hd, tl.as_ref())) } else { None }
   } }
fn test() -> (List<i64>,List<i64>) {
   let a = Rc::new(Cons(5, Rc::new(Nil)));
   let b = Cons(3, Rc::clone(&a));
   let c = Cons(4, Rc::clone(&a));
   (b,c)   }
fn main() {
   let (l1, l2) = test();   { let _t = l1; }
   println!("{}", l2.unfold().unwrap().1.unfold().unwrap().0)   }
```

RefCell type

Use RefCell type to replace static mutation-xor-sharing checking with dynamic checking.

<Problem>

Want to mutate the shared list in the previous example.

RefCell type

```
use std::rc::Rc; use std::cell::{RefCell, Ref, RefMut};
enum List<T> {
    Cons(T, Rc<RefCell<List<T>>>),
    Nil,
}
use crate::List::{Cons, Nil};
impl <T> List<T> {
    fn unfold<'a>(&'a self) -> Option<(&'a T, Ref<'a, List<T>>)> {
        if let Cons(hd,tl) = self { Some((hd, tl.borrow())) } else { None }
                                                        // tl.as_ref().borrow()
    }
    fn unfold_mut<'a>(&'a mut self) -> Option<(&'a mut T, RefMut<'a,
List<T>>)> {
        if let Cons(hd,tl) = self { Some((hd,tl.borrow_mut())) } else { None }
    } }
```

RefCell type

```
fn test() -> (List<i64>,List<i64>) {
    let a = Rc::new(RefCell::new(Cons(5, Rc::new(RefCell::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
    (b,c)
}
fn main() {
    let (mut l1, l2) = test();
    {
        let mut r = l1.unfold_mut().unwrap().1;
        *r = Cons(42, Rc::new(RefCell::new(Nil)));
    }
    println!("{}", l2.unfold().unwrap().1.unfold().unwrap().0)
}
```

Type class and Closure

Type class

```
trait Describable {
  fn describe(&self) -> String;
}
struct Book { title: String, author: String, }
impl Describable for Book {
    fn describe(&self) -> String {
        format!("{} by {}", self.title, self.author)
    } }

//fn print_desc(a: &impl Describable) -> ()
fn print_desc<T>(a: &T) -> () where T : Describable
{ println!("{}", a.describe()) }
fn main() {
    print_desc(&Book{title: "PP".to_string(), author: "Gil Hur".to_string()})
}
```

Dyn: dynamic dispatch

```
impl Describable for i64 {
    fn describe(&self) -> String {
        format!("64-bit-signed-integer: {}", *self)
    }
  }

fn main() {
    let book = Book{title: "PP".to_string(), author: "Gil Hur".to_string()};
    let ds : Vec<Box<dyn Describable>> =
        vec![Box::new(42), Box::new(book)];
    for d in &ds {
        println!("{}", d.describe())
        // cannot use print_desc : why?
    }
}
```

Closure type: Fn

```
fn test1<F>(f: F) -> usize
where F: Fn(usize)->usize {
    f(10) + f(20)
}
fn main() {
    let mut s = "abc".to_string();
    println!("{}", test1(|n|n+s.len()));
}
```

Closure type: FnMut

```
fn test2<F>(mut f: F)
where F: FnMut(&str)->() {
    f("gil");
    f("hur");
}
fn main() {
    let mut s = "abc".to_string();
    println!("{}", test1(|n|n+s.len()));
    test2(|x|s.push_str(x));
    println!("{}", test1(|n|n+s.len()));
}
```

Closure type: FnOnce

```
fn test3<F>(f: F)->String
where F: FnOnce(&str)->String {
    f("test")
}
fn main() {
    let mut s = "abc".to_string();
    println!("{}", test1(|n|n+s.len()));
    test2(|x|s.push_str(x));
    println!("{}", test1(|n|n+s.len()));
    let s2 = test3(|x|{s.push_str(x); s});
    println!("{}", s2);
}
```

Closure type: move

```
fn main() {
    let clo;
    {
        let mut s = "abc".to_string();
        println!("{}", test1(|n|n+s.len()));
        test2(|x|s.push_str(x));
        println!("{}", test1(|n|n+s.len()));
        let s2 = test3(|x|{s.push_str(x); s});
        println!("{}", s2);
        clo = move |n|n+s2.len();
    }
    println!("{}", test1(clo));
}
```

Thanks for your hard work!