Functional Collections

06016415 Functional Programming

IT KMITL
พระจอมเกล้าลาดกระบัง

Outline

- Immutable vs Mutable Variable
- Sequence, Map, Set
- Functional Lists

Produces two different strings from the same input, making it impure

- Local variables จะไม่เปลี่ยนแปลงค่า
- Global variables สามารถเปลี่ยนแปลงค่า เฉพาะเมื่อถูกอ้างอิง (only references)

```
var prompt = "> "
def format(msg: String): String = prompt + msg

format("command") // "> command"
prompt = "% "     // change the prompt
format("command") // "% command"
```

This function has the side effect of modifying variable lastID

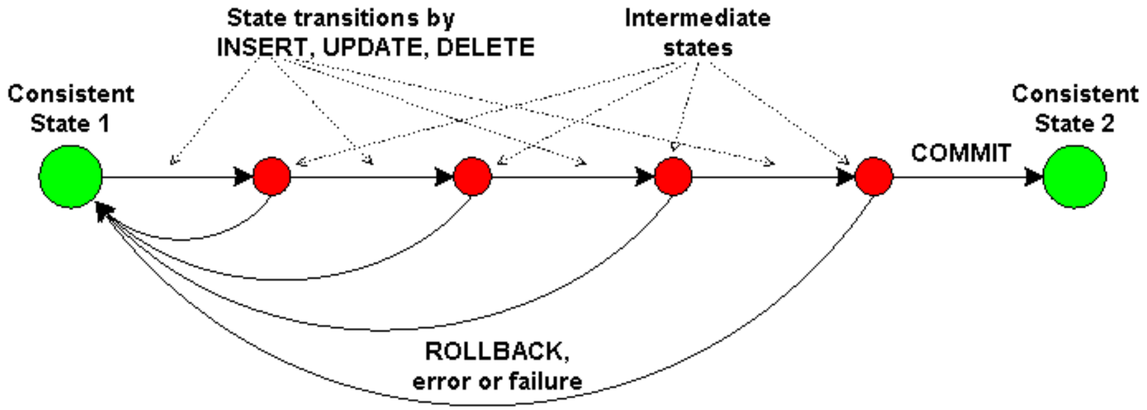
```
var lastID = 0
def uniqueName(prefix: String): String =
    lastID += 1
    prefix + lastID

uniqueName("user-") // "user-1"
uniqueName("user-") // "user-2"
```

Immutable vs Mutable Variable

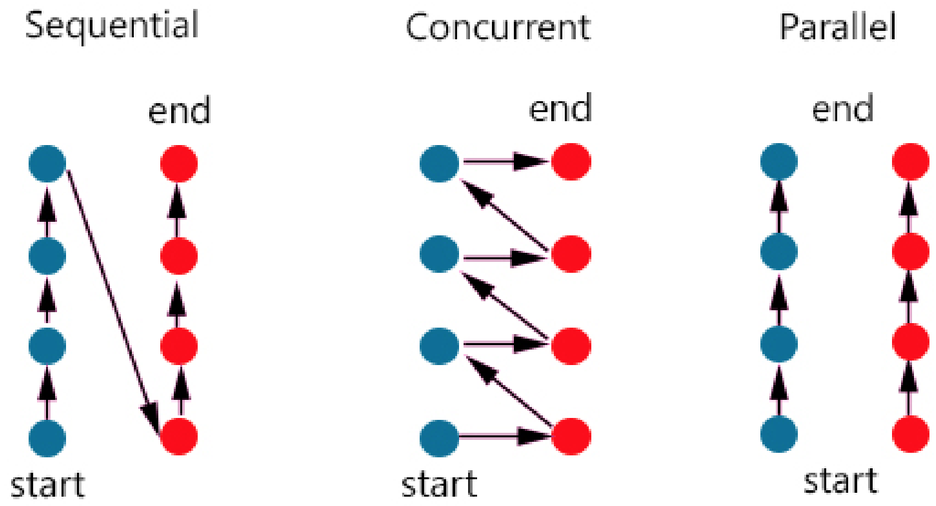
- **FP** มาจากแนวคิด **Mathematical Functions** ซึ่งจะต้องไม่เปลี่ยนแปลงค่า **(Immutable)**
- ตัวแปรแบบ Immutable จะทำให้การสร้าง Pure Function เกิดขึ้นได้
  - Pure function จะต้องได้รับค่าผลลัพธ์เดิมเสมอ
  - Output depends on Input
  - ฟังก์ชันที่ไม่มี side effects.

- Immutable variables หรือ ตัวแปรไม่เปลี่ยนค่า อาจทำให้เราสับสน กับตัวแปร (Mutable variables) ที่เคยรู้จัก ซึ่งมีไว้เพื่อเปลี่ยนแปลงค่า แล้วนำไปคำนวณ

- **Immutable variables คือ Variables That Aren’t**



Example of immutability : Database Transactions



Sequential VS Concurrent VS Parallel

- ประโยชน์ของ Immutability คือช่วยแก้ปัญหา concurrency คือปัญหาการเข้าถึงข้อมูลที่ร่วมกัน แล้วเกิดความไม่แน่นอน หรือคืนค่าผลลัพธ์ผิดพลาด (ตัวอย่าง โจทย์ big data เป็นต้น)

Immutable vs Mutable Collections

Immutable Collections (ค่าเริ่มต้นของ Scala) กับ Mutable Collections ใน Scala 3 แบ่งเป็น 2 แพ็กเกจหลักชัดเจน:

1.scala.collection.immutable (Default):

- แก้ไขไม่ได้ ถ้าจะแก้ต้องสร้างใหม่ (ปลอดภัย, Thread-safe)

2.scala.collection.mutable (Optional):

- แก้ไขค่าใน Memory เดิมได้ (เหมือน Java/Python, เร็วในบางกรณีแต่เสี่ยง)

Immutable Collections

ใน Scala (และ FP) เราใช้ Collection ที่แก้ไขไม่ได้ (Immutable) เป็นหลัก เมื่อมีการเพิ่ม/ลบข้อมูล เราจะได้ Collection ใหม่ เสมอ

Collection	คุณสมบัติเด่น	เหมาะสำหรับ
List	Linked List (Head :: Tail)	ข้อมูลที่เน้นทำกับตัวหน้าสุด (Recursion), Prepend เร็ว
Vector	Tree-based Array	ข้อมูลขนาดใหญ่, Random Access เร็ว, Append เร็ว (แนะนำให้ใช้แทน Array)
Set	Unique Elements	เก็บข้อมูลที่ไม่ซ้ำกัน, เช็คว่า "มีของชิ้นนี้ไหม" ได้เร็ว
Map	Key-Value Pairs	Dictionary, Lookup Table

- **Scala**
  - Hybrid object-functional language
  - Not require functions to be pure
  - Write your code this way whenever possible

Variable Declarations

```
scala> val str = Seq("STILL", "MORE", "HELLO", "WORLD")
val str: Seq[String] = List(STILL, MORE, HELLO, WORLD)

scala> str.map(_.toLowerCase)
val res0: Seq[String] = List(still, more, hello, world)

scala> str.map(_.toUpperCase)
val res1: Seq[String] = List(STILL, MORE, HELLO, WORLD)
```

```
scala> val seq: Seq[String] = Seq("This", "is", "Scala")
val seq: Seq[String] = List(This, is, Scala)

scala> val array: Array[String] = Array("This", "is", "Scala")
val array: Array[String] = Array(This, is, Scala)
```

Human class with an immutable name, but a mutable age

```
//class
class Human(val name: String, var age: Int)
val p = Human("Dean Wampler", 29)
```

Ranges

```
scala> 1 to 10
val res7: scala.collection.immutable.Range.Inclusive = Range 1 to 10

scala> 1 until 10
val res8: Range = Range 1 until 10

scala> 1 to 10 by 3
val res9: Range = Range 1 to 10 by 3

scala> (1 to 10 by 3).foreach(println)
1
4
7
10

scala> ('a' to 'g' by 3).foreach(println)
a
d
g
```

Tuples

```
scala> val tup = ("Hello", 1, 2.3)
val tup: (String, Int, Double) = (Hello,1,2.3)

scala> val tup2: (String, Int, Double) = ("World", 4, 5.6)
val tup2: (String, Int, Double) = (World,4,5.6)

scala> (tup._1, tup(0))
val res10: (String, String) = (Hello,Hello)

scala>  (tup._2, tup(1))
val res11: (Int, Int) = (1,1)

scala> (tup._3, tup(2))
val res12: (Double, Double) = (2.3,2.3)

scala> (tup._4, tup(3))
-- [E008] Not Found Error: ----------------------------------------------------
1 |(tup._4, tup(3))
  | ^^^^^^
  | value _4 is not a member of (String, Int, Double) - did you mean tup._1?
-- Error: ---------------------------------------------------------------------
1 |(tup._4, tup(3))
  |            ^
  |        Match type reduction failed since selector EmptyTuple.type
  |        matches none of the cases
  |
  |            case x *: xs => (0 : Int) match {
  |          case (0 : Int) => x
  |          case scala.compiletime.ops.int.S[n1] => scala.Tuple.Elem[xs, n1]
  |        }
2 errors found
```

Anonymous Functions

```
scala> var factor = 2
var factor: Int = 2

scala> val multiplier = (i: Int) => i * factor
val multiplier: Int => Int = Lambda$6961/1069660924@695ab908

scala> val result1 =(1 to 10).filter(_ % 2 == 0).map(multiplier).reduce(_ * _)
val result1: Int = 122880

scala> factor = 3
factor: Int = 3

scala> val result2 = (1 to 10).filter(_ % 2 == 0).map(multiplier).reduce(_ * _)
val result2: Int = 933120
```

Anonymous Functions

```
def mult: Int => Int =
  val factor = 2
  (i :Int ) => i *factor
```

```
scala> val result3= (1 to 10).filter(_ % 2 == 0).map(mult).reduce(_ * _)
val result3: Int = 122880
```

```scala
def mult: Int => Int =
  val factor = 2
  (i :Int ) => i *factor

def multiplier(i: Int, factor : Int): Int =
  i * factor

var factor2 = 2
def multiplier2(i: Int) = i * factor2
```

```
scala> val result3= (1 to 10).filter(_ % 2 == 0).map(mult).reduce(_ * _)
val result3: Int = 122880

scala> val result3= (1 to 10).filter(_ % 2 == 0).map(multiplier2).reduce(_ * _)
val result3: Int = 122880

scala> val result3= (1 to 10).filter(_ % 2 == 0).map(multiplier).reduce(_ * _)
val result3: Int = 122880
```

LazyList in the standard library

Like many of the other data structures we've seen so far,
LazyList exists in the Scala standard library (see the API at https://mng.bz/M00D).

```
scala> val natNums = LazyList.from(0)
val natNums: LazyList[Int] = LazyList(<not computed>)

scala> natNums.take(100).toList
val res0: List[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83,
 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99)
```

Vector vs. ArrayBuffer

- Vector (Immutable): เป็นจุดเด่นของ Scala Collection ออกแบบด้วยโครงสร้าง Bit-mapped Trie (ต้นไม้ที่มีกิ่งก้านกว้างมาก)

- ArrayBuffer (Mutable): เหมือน ArrayList ใน Java หรือ list ใน Python คือ Array ที่ยืดหดขนาดได้เอง

```
import scala.collection.mutable.ArrayBuffer

val vec1 = Vector(1, 2, 3)
val vec2 = vec1.updated(0, 99) // แก้ตัวที่ 0 เป็น 99 (ได้ Vector ใหม่)

val arrBuf = ArrayBuffer(1, 2, 3)
arrBuf(0) = 99 // แก้ค่าใน Memory เดิม
arrBuf += 4    // ขยายขนาด Array อัตโนมัติ
```

- Functional programming จะเน้นการใช้ ความรู้พื้นฐานด้าน Data Structures and Algorithms
- Add, Changes, Remove โดยไม่แก้ไข โครงสร้างข้อมูลเดิมในโปรแกรม

Sequences

- Data Structures โดยทั่วไปมักกำหนดให้มีโครงสร้างแบบ Sequential (ตามลำดับ)
- Array, List, Seq, Vector

```
scala> Seq.apply()
val res5: Seq[Nothing] = List()
```

Append and Prepend on Seq

Method	Description	Example
:+	append 1 item	oldSeq :+ e
++	append N items	oldSeq ++ newSeq
+:	prepend 1 item	e +: oldSeq
++:	prepend N items	newSeq ++: oldSeq

```
scala> val seq1 = Seq("Programming", "Scala")
val seq1: Seq[String] = List(Programming, Scala)
```

```
scala> val seq2 = "Programming" +: "Scala" +: Nil
val seq2: List[String] = List(Programming, Scala)
```

Nil is a subtype of List that is a convenient object when an empty list is required.

```
scala> val seq3 = "People" +: "should" +: "read" +: seq1
val seq3: Seq[String] = List(People, should, read, Programming, Scala)

scala> seq3.head
val res6: String = People

scala> seq3.tail
val res7: Seq[String] = List(should, read, Programming, Scala)
```

+: (prepend), and :+ (append)

```
scala> seq
val res1: Seq[String] = List(This, is, Scala)

scala> array
val res2: Array[String] = Array(This, is, Scala)
```

```
scala> array = Array("Bad!")
-- [E052] Type Error: ----------------------------------------------
1 |array = Array("Bad!")
  |^^^^^^^^^^^^^^^^^^^^
  |Reassignment to val array
  |
  | longer explanation available when compiling with `-explain`
1 error found

scala> array(1) = "still is"

scala> seq(1)
val res3: String = is

scala> array
val res4: Array[String] = Array(This, still is, Scala)

scala> var seq2: Seq[String] = Seq("This", "is", "Scala")
var seq2: Seq[String] = List(This, is, Scala)

scala> seq2 = Seq("No", "longer", "Scala")
seq2: Seq[String] = List(No, longer, Scala)
```

เพิ่มเครื่องหมาย () เพื่อจัดลำดับในการสร้างตัวแปรได้

```
scala> val seq2b = ("Programming" +: ("Scala" +: (Nil)))
val seq2b: List[String] = List(Programming, Scala)
```

Vector คุณสมบัติเป็นโครงสร้างแบบ **dynamic** มีลักษณะการจัดเก็บข้อมูลคล้ายกับ **Array** แต่ **vector** มีความหยืดหยุ่น และมีประสิทธิภาพมากกว่า

```
scala> val vect1 = Vector("Programming","Scala")
val vect1: Vector[String] = Vector(Programming, Scala)

scala> val vect2 = "People" +: "should" +: "read" +: Vector.empty
val vect2: Vector[String] = Vector(People, should, read)

scala> val vect3 = "People" +: "should" +: "read" +: vect1
val vect3: Vector[String] = Vector(People, should, read, Programming, Scala)

scala> val vect4 = Vector.empty :+ "People" :+ "should" :+ "read"
val vect4: Vector[String] = Vector(People, should, read)
```

+: (prepend), and :+ (append)

```
scala> vect3.head
val res8: String = People

scala> vect3.tail
val res9: Vector[String] = Vector(should, read, Programming, Scala)

scala> val seq1 = Seq("Programming", "Scala")
val seq1: Seq[String] = List(Programming, Scala)

scala> val vect5 = seq1.toVector
val vect5: Vector[String] = Vector(Programming, Scala)
```

Maps

- Maps เป็น โครงสร้างข้อมูลพื้นฐานทั่วไป เพื่อใช้จับคู่ระหว่าง Key กับ Value
- Key ต้องเป็น unique

```
scala> Map.apply()
val res4: Map[Nothing, Nothing] = Map()
```

```
scala> map.apply()
-- [E006] Not Found Error: ----------------------------------------------------
1 |map.apply()
  |^^^
  |Not found: map
  |
  | longer explanation available when compiling with `-explain`
1 error found
```

- คล้ายกับ Seq แต่มีลักษณะการใช้งานต่างกัน

```scala
val stateCapitals = Map(
  "Alabama" -> "Montgomery",
  "Alaska" -> "Juneau",
  "Wyoming" -> "Cheyenne")

val stateCapitals2a = stateCapitals + ("Virginia" -> "Richmond")
val stateCapitals2b = stateCapitals + ("Alabama" -> "MONTGOMERY")
val stateCapitals2c = stateCapitals ++ Seq("Virginia" -> "Richmond", "Illinois" -> "Springfield")
```

```
scala> stateCapitals
val res0: Map[String, String] = Map(Alabama -> Montgomery, Alaska -> Juneau, Wyoming -> Cheyenne)

scala> stateCapitals2a
val res1: Map[String, String] = Map(Alabama -> Montgomery, Alaska -> Juneau, Wyoming -> Cheyenne, Virginia -> Richmond)

scala> stateCapitals2b
val res2: Map[String, String] = Map(Alabama -> MONTGOMERY, Alaska -> Juneau, Wyoming -> Cheyenne)

scala> stateCapitals2c
val res3: Map[String, String] = HashMap(Alaska -> Juneau, Illinois -> Springfield, Wyoming -> Cheyenne, Virginia -> Richmond, Alabama -> Montgomery)
```

Map vs. mutable.Map

- Map (Immutable): เก็บข้อมูลแบบ Key-Value เมื่อมีการเพิ่ม/ลบ Key จะได้ Map ใบใหม่ แต่ข้อมูลข้างในส่วนใหญ่ยังใช้ร่วมกันกับของเก่า (Hash Array Mapped Trie)

- mutable.Map (Mutable): เก็บข้อมูลแบบ Hash Table ปกติ แก้ไขค่าได้ทันทีScala

```
import scala.collection.mutable

val users = Map("id1" -> "Somchai")
val users2 = users + ("id2" -> "Somsri") // users เดิมยังอยู่ครบ

val mUsers = mutable.Map("id1" -> "Somchai")
mUsers("id1") = "Sombat" // เปลี่ยนชื่อ Somchai เป็น Sombat ในที่เดิม
```

Sets

- Sets คือ โครงสร้างข้อมูลที่ไม่จัดลำดับ
- Sets คล้ายกับ Maps Key คือ แต่ละ elements ต้องเป็น unique

```
scala> Set.apply()
val res5: Set[Nothing] = Set()
```

```
scala> val states = Set("Alabama", "Alaska", "Wyoming")
val states: Set[String] = Set(Alabama, Alaska, Wyoming)

scala> val states2 = states + "Virginia"
val states2: Set[String] = Set(Alabama, Alaska, Wyoming, Virginia)

scala> val states3 = states ++ Seq("New York", "Illinois", "Alaska")
val states3: Set[String] = HashSet(Alaska, Alabama, New York, Illinois, Wyoming)
```

The Scala standard library implements both mutable and immutable sets.

```
scala> var set = Set("A","B")
//var set: Set[String] = Set(A, B)


scala> set =set+"C"
//set: Set[String] = Set(A, B, C)

scala> set =set+"D"
//set: Set[String] = Set(A, B, C, D)



scala> val set = Set("A","B")
//val set: Set[String] = Set(A, B)

scala> set + "C"
//val res1: Set[String] = Set(A, B, C)

scala> set
//val res2: Set[String] = Set(A, B)
```

Set vs. mutable.Set

- เหมือนกับ Map ทุกประการ แต่เก็บแค่ Key (ห้ามซ้ำ)

```
val s1 = Set(1, 2, 3)
val s2 = s1 + 1 // s2 ก็ยังเป็น Set(1, 2, 3) เพราะ 1 ซ้ำ

val ms = mutable.Set(1, 2)
ms += 3 // แก้ไข object เดิม
```

Functional Lists

- Functional list เป็นโครงสร้างข้อมูลพื้นฐาน ใช้โดยทั่วไป
- Empty list คือ ไม่มี elements หรือ nil
- Head คือ ค่าแรก ใน list
- Tail คือ ค่าส่วนที่เหลือ หรือส่วนท้ายใน list
- ตัวดำเนินการ หรือ “cons.” เช่น :: , +: และ :+ เป็นต้น

```
scala> 1 :: 2 :: 3 :: Nil
val res0: List[Int] = List(1, 2, 3)
```

List vs. ListBuffer

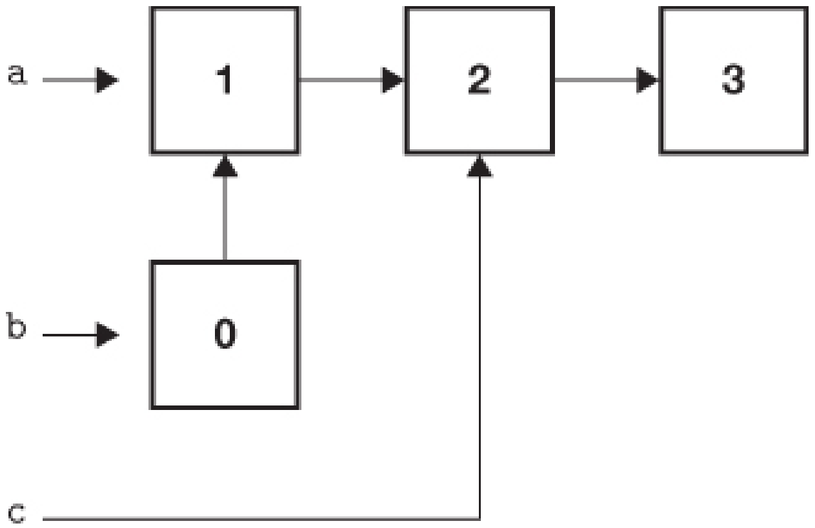
- List (Immutable): เป็นโครงสร้างแบบ Linked List (Head :: Tail) เหมาะสำหรับการทำงานแบบ Recursion หรือเอาข้อมูลใส่ข้างหน้า (Prepend)

- ListBuffer (Mutable): ใช้เมื่อเราต้องการ "สร้าง List" ใน Loop หรือต่อท้ายข้อมูลเรื่อยๆ อย่างมีประสิทธิภาพ เมื่อเสร็จแล้วค่อยแปลงกลับเป็น List

```
import scala.collection.mutable.ListBuffer

val list1 = List(1, 2, 3)
val list2 = 0 :: list1 // สร้างใหม่โดยเอา 0 มาต่อหน้า (list1 ไม่เปลี่ยน)

val buf = ListBuffer[Int]()
buf += 1  // แก้ไขที่เดิม
buf += 2
val result: List[Int] = buf.toList // แปลงกลับเป็น Immutable เพื่อความปลอดภัย
```

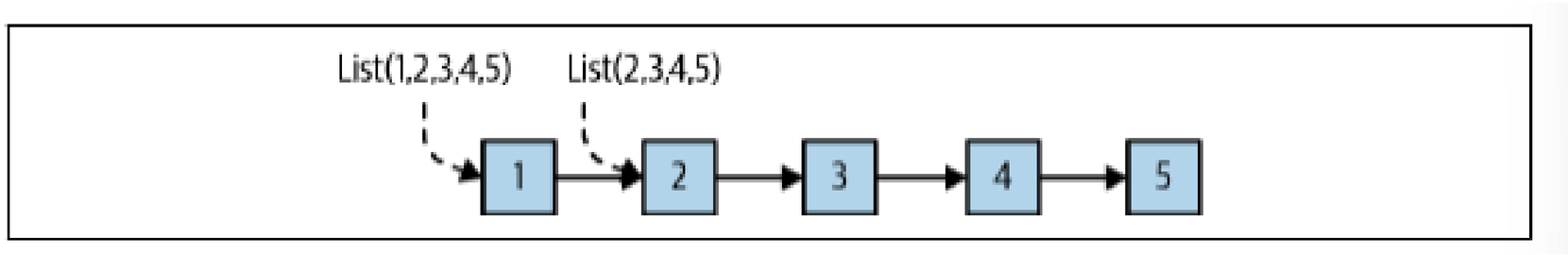
Structural Sharing

Functional lists memory sharing

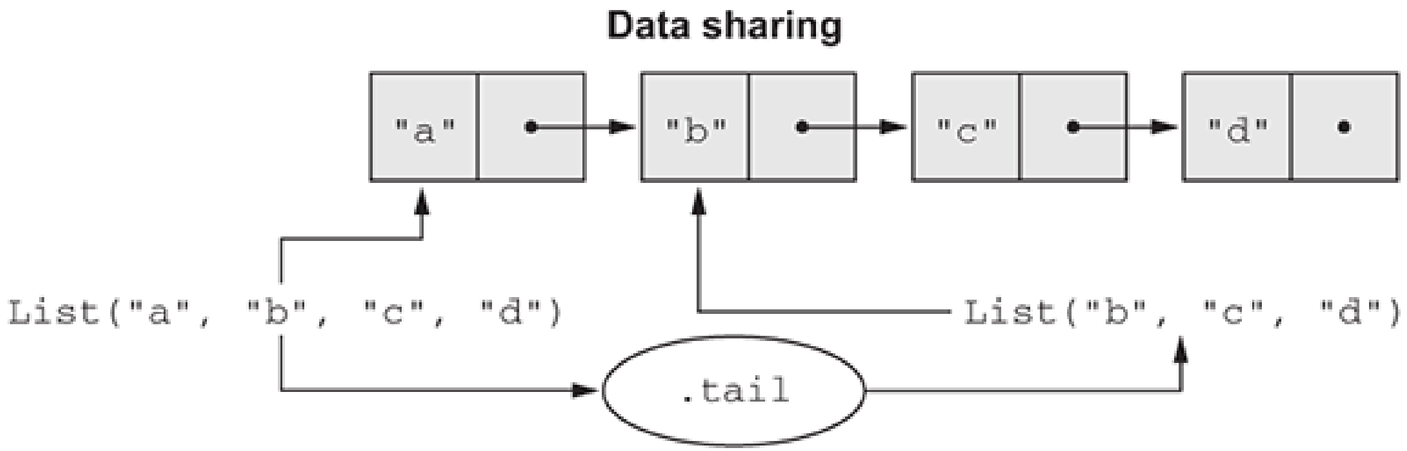



```
scala> val a = List(1, 2, 3)
val a: List[Int] = List(1, 2, 3)

scala> val b = 0 :: a
val b: List[Int] = List(0, 1, 2, 3)

scala> val c = a.tail
val c: List[Int] = List(2, 3)
```

Lists are immutable

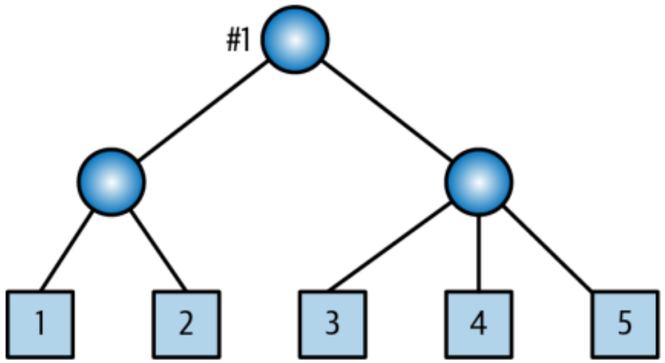


In functional data structures, sharing a structure to minimize the cost of making copies.

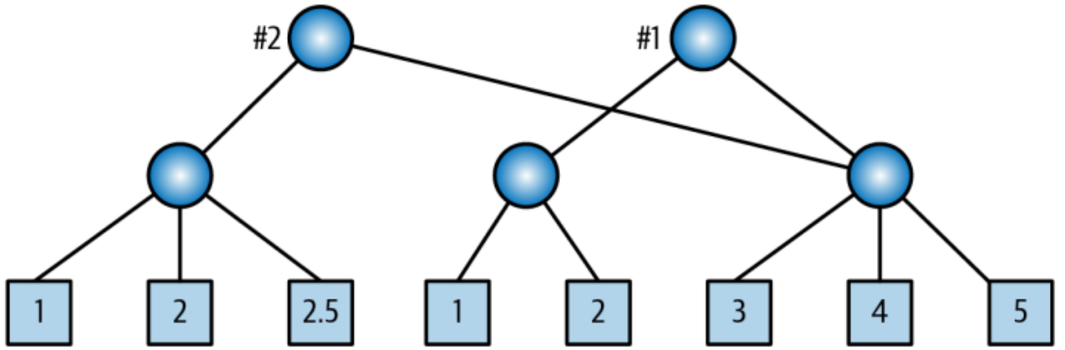


Both lists share the same data in memory. `.tail` does not modify the original list; it simply references the tail of the original list. Defensive copying is not needed because the list is immutable.

Making copies of functional collections



A Vector represented as a tree



Element insertion

Summary

- List (Immutable): เป็นโครงสร้างแบบ Linked List (Head :: Tail) เหมาะสำหรับการทำงานแบบ Recursion หรือเอาข้อมูลใส่ข้างหน้า (Prepend)
- Start with Immutable: เริ่มต้นให้ใช้ List, Vector, Map (แบบ Default) เสมอ เพื่อความปลอดภัยและลด Side Effect
- (Optional)Local Mutation: ถ้าในฟังก์ชันมีการวนลูปซับซ้อน หรือต้องสร้าง List ขนาดใหญ่ ให้ใช้ ListBuffer หรือ ArrayBuffer ภายในฟังก์ชันนั้น แล้วตอน return ให้แปลงกลับเป็น Immutable (.toList, .toVector)