

Strictness and Laziness

06016415 Functional Programming

- Evaluation Strategies
 - Strict / Non-strict Evaluation
- Lambda Calculus Example
- Strict and Non-strict Functions

- In programming language, an **evaluation strategy** is a set of rules for evaluating expressions.
- Example

$$(1-x)(1+x)$$

when x is very close to 1 might be better evaluated as $1-2x-x^2$.

$$f(x) = \frac{a(x)}{b(x)}$$

Our lazy value is $a(x)$, and this is the point of a lazy variable.

Where evaluation strategies are important, say in numerical computation, we need to know a lot about how the compiler works.

- **Strict evaluation** means to evaluate all the arguments to a function before evaluating the function.
 - *When we compute a function we usually assume that all arguments have a value before the function evaluation begins*
- **Non-strict evaluation** arguments need not be evaluated until they are actually required. (**call-by-need**)
 - *Lazy evaluation is a form of non-strict evaluation in which arguments are not evaluated until required.*

Table of evaluation strategies

Evaluation strategy	Representative Languages	Year first introduced
Call by value	ALGOL, C, Scheme, MATLAB	1960
Call by name	ALGOL 60, Simula	1960
Call by unification	Prolog	1965
Call by need	SASL, Haskell, R	1971
Call by sharing	CLU, Java, Python, Ruby, Julia	1974
Call by reference parameters	C++, PHP, C#, Visual Basic .NET	1985
Call by reference to const	C++, C	1985

Laziness in the real world



*Find odd numbers of
an array of numbers
from 1 to 6.*

Lambda Calculus Example

- In the λ -calculus, all reduction orders that terminate give the same result.
- Changing the evaluation strategy only changes whether the program terminates, not the final result of the computation.

```
(define X 2)
```

```
(define Y 3)
```

```
(if test-expr then-expr else-expr)
```

```
(if (< X Y)
    (+ X X)
    (* Y Y)
  )
```

```
(list (< X Y)
      (print "true")
      (print "false")
    )
```

```
(or (and condition consequent) alternate)
```

```
(or (and (< X Y)
          (+ X X)
        )
    (* Y Y)
  )
```

```
(or (and (< X Y)
          (print "true")
        )
    (print "false")
  )
```

- **Nonstrictness** is a property of a function. To say that a function is nonstrict just means the function may choose not to evaluate one or more of its arguments. In contrast, a *strict* function always evaluates its arguments.
- **Strict functions** are the norm in most programming languages, and indeed, most languages only support functions that expect their arguments fully evaluated.

```
def square(x: Double): Double = x * x
```

```
def nonStrictSquare(x: => Double): Double = x * x
```

A nonstrict if function

```
def if1[A](cond: Boolean, onTrue: () => A, onFalse: () => A): A =  
  if cond then onTrue() else onFalse()  
  
def if2[A](cond: Boolean, onTrue: => A, onFalse: => A): A =  
  if (cond) onTrue else onFalse
```

The **onTrue** and **onFalse** arguments use some new syntax we have not encountered yet: the type `() => A`. A value of type `() => A` is a function that accepts zero arguments and returns an `A`.

```
scala> if2(false, sys.error("fail"), 3)  
val res0: Int = 3  
  
scala> if2(true, sys.error("fail"), 3)  
java.lang.RuntimeException: fail  
  at scala.sys.package$.error(package.scala:27)  
  at rs$line$3$$init$$anonfun$1(rs$line$3:1)  
  at Main$package$.if2(Main.scala:11)  
  ... 62 elided
```

```
scala> if2(true, 1, 3)  
val res2: Int = 1  
  
scala> if1(true, 1, 3)  
-- [E007] Type Mismatch Error:  
1 | if1(true, 1, 3)  
   | ^  
   | Found:  (1 : Int)  
   | Required: () => Any  
   | longer explanation available when compiling with `~-explain`  
-- [E007] Type Mismatch Error:  
1 | if1(true, 1, 3)  
1 | if1(true, 1, 3)  
   | ^  
   | Found:  (1 : Int)  
   | Required: () => Any  
   | longer explanation available when compiling with `~-explain`  
-- [E007] Type Mismatch Error:  
1 | if1(true, 1, 3)  
   | ^  
   | Found:  (3 : Int)  
   | Required: () => Any  
   | longer explanation available when compiling with `~-explain`  
2 errors found
```

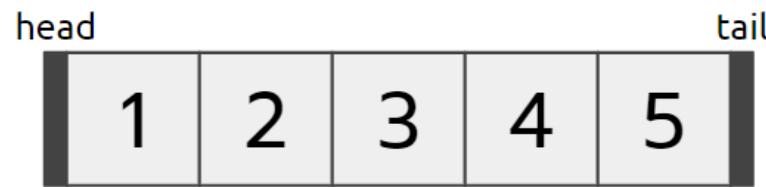
```
scala> List(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).map(_ * 3)
val res0: List[Int] = List(36, 42)

scala> List(1,2,3,4).map(_ + 10)
val res1: List[Int] = List(11, 12, 13, 14)

scala> List(1,2,3,4).map(_ + 10).filter(_ % 2 == 0)
val res2: List[Int] = List(12, 14)

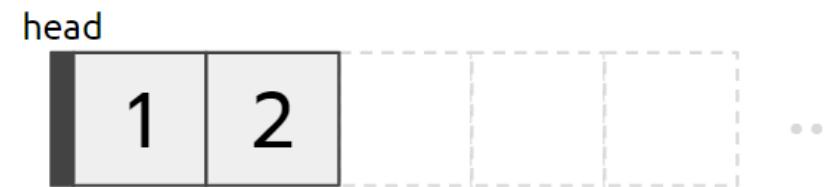
scala> List(1,2,3,4).map(_ + 10).filter(_ % 2 == 0).map(_ * 3)
val res3: List[Int] = List(36, 42)
```

evenNumbers = [1,2..1000]

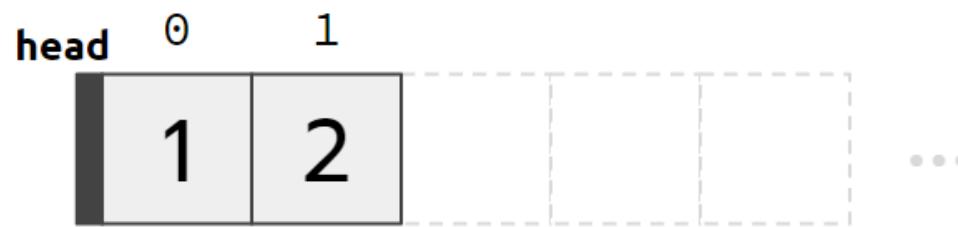


evenNumbers = [1,2..]

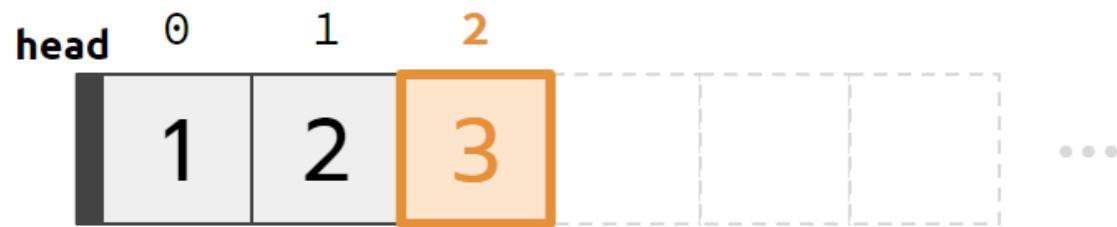
static



lazy



access [2]



access [4]



- **LazyLists** are similar to lists, but their elements are evaluated only on demand.
- It is "lazy" because it computes its elements only when they are needed.

LazyList exists in the Scala standard library, and it is a data structure designed for lazy evaluation. This means that elements of a **LazyList are not calculated until they are accessed**, making it efficient for working with large or infinite sequences of data.

- A **LazyList** is like a regular list, but it doesn't compute its elements until you ask for them.
- It is **immutable** (once created, it cannot be changed).
- Useful for infinite sequences or large datasets where computing everything upfront would be inefficient.

```
@main def lazyListExample(): Unit = {
    // Create a LazyList
    val nums = LazyList(1, 2, 3, 4, 5)

    // Access the first element (head)
    println(nums.head)          // Output: 1

    // Access the next element (tail's head)
    println(nums.tail.head)     // Output: 2

    // Convert to a regular List (forces all elements to be computed)
    println(nums.toList)        // Output: List(1, 2, 3, 4, 5)
}
```

```
@main def infiniteLazyList(): Unit = {
    // Create an infinite LazyList of numbers starting from 1
    val infiniteNums = LazyList.iterate(1)(_ + 1)

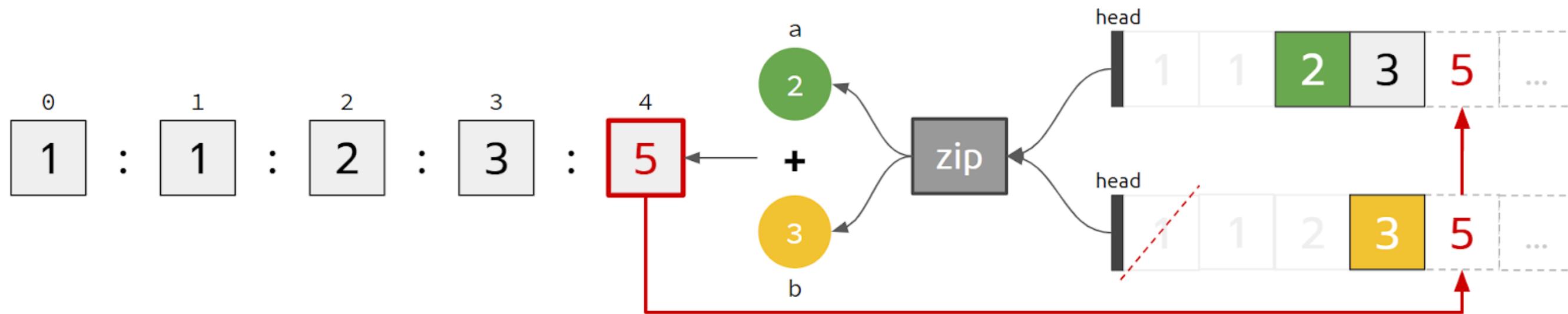
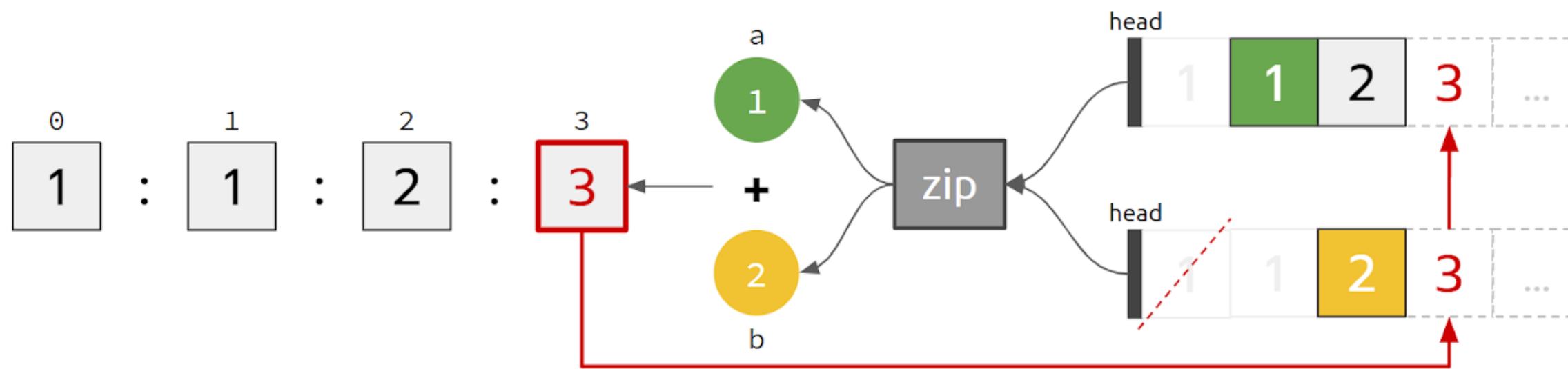
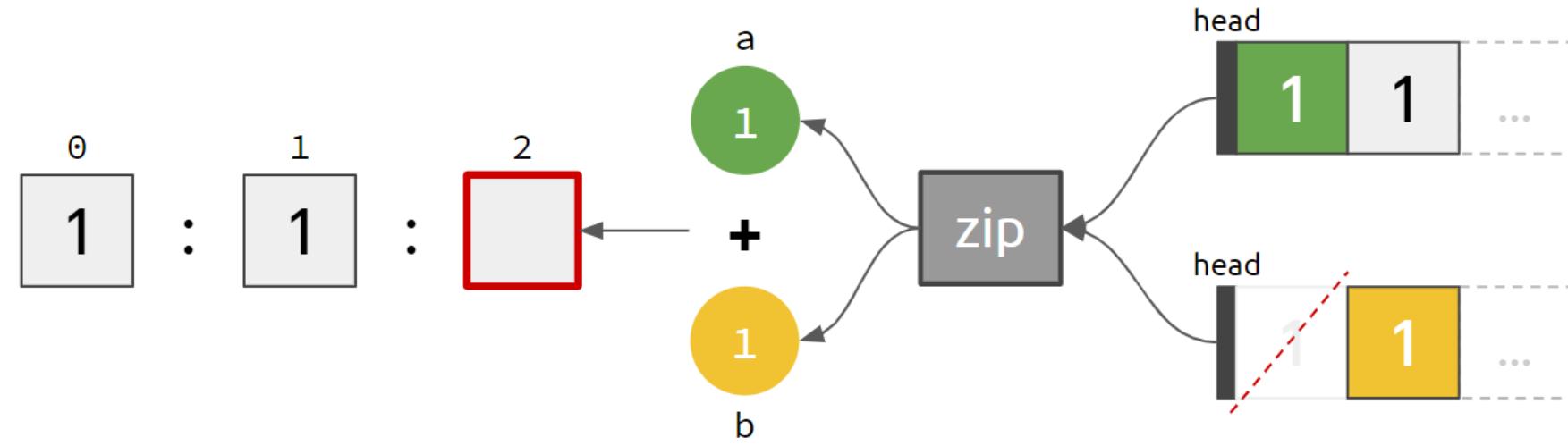
    // Take the first 5 elements
    println(infiniteNums.take(5).toList) // Output: List(1, 2, 3, 4, 5)

    // Filter even numbers and take the first 5
    val evenNums = infiniteNums.filter(_ % 2 == 0)
    println(evenNums.take(5).toList)    // Output: List(2, 4, 6, 8, 10)
}
```

```
@main def fibonacciLazyListSimple(): Unit = {
    // Define the Fibonacci sequence using LazyList
    lazy val fibs: LazyList[BigInt] = LazyList.iterate((0, 1)) { case (a, b) =>
        (b, a + b) }.map(_._1)

    // Take the first 10 Fibonacci numbers
    println(fibs.take(10).toList) // Output: List(0, 1, 1, 2, 3, 5, 8, 13, 21,
34)

    // Show Fibonacci at 100
    println(fibs(100)) // Output: 354224848179261915075
}
```



- **Group work**
- **Discuss on**
 - **Laziness of AND / OR / NOT operations**
 - **LazyList in Scala**
 - **Problem of Laziness , map() and filter() in Scala**
- **Summarize and Show example code**