

CSC 347 - Concepts of Programming Languages

Functional Programming with Lists

Instructor: James Riely



Learning Objectives

- ❓ What are common ways of processing collections functionally?
 - Express and interpret collections processing in a functional style
 - Compare state of computation in iteration vs. recursion
 - Express and interpret collections processing with list comprehensions
 - Infer result types of for comprehensions



Programs are Proofs

- Function: $f(n) = 2 + 2 + \dots + 2$ (n times)
- Theorem: for all natural numbers n , $f(n) = n \cdot 2$
- Proof by induction:
 - **Base case** ($n = 0$): show $0 \cdot 2 = 0$ ✓
 - **Induction hypotheses (IH)**: assume $f(n - 1) = (n - 1) \cdot 2$
 - **Inductive case**
 - show $f(n) = n \cdot 2$
 - expands to $f(n - 1) + 2 = n \cdot 2$
 - by IH $(n - 1) \cdot 2 + 2 = n \cdot 2$
 - by arithmetic $n \cdot 2 = n \cdot 2$ ✓
- Implement proof in Scala as a recursive function
- This is a recursive definition
- Curry-Howard Correspondence

```
1 def f(n: Int) : Int = n match
2   case 0 => 0
3   case n => f(n-1) + 2
4 end f
```



Exercise: Print Every Element of a List

? Express in a functional style with pattern matching

```
1 def printList (xs:List[Int]) : Unit =  
2  
3  
4  
5  
6  
7  
8 val xs = List(11,21,31)  
9 printList (xs)
```



Exercise: Print Every Element of a List

? Visit all elements of a list

```
1 def printList (xs:List[Int]) : Unit = xs match
2   case Nil      => ()
3   case y::ys =>
4
5     printList (ys)
6
7
8 val xs = List(11,21,31)
9 printList (xs)
```



Exercise: Print Every Element of a List

? Print an element when visiting

```
1 def printList (xs:List[Int]) : Unit = xs match
2   case Nil    => ()
3   case y::ys =>
4     println (y)
5     printList (ys)
6
7
8 val xs = List(11,21,31)
9 printList (xs)
```



Exercise: Print Every Element of a List

? Format every element before printing

```
1 def printList (xs:List[Int]) : Unit = xs match
2   case Nil      => ()
3   case y::ys =>
4     println (s"y=$y")
5     printList (ys)
6
7
8 val xs = List(11,21,31)
9 printList (xs)
```



List Operation: Foreach

💡 Generalize the idea of processing every element

```
1 def foreach (xs:List[Int], f:Int=>Unit) : Unit = xs match
2   case Nil    => ()
3   case y::ys =>
4     f (y)
5     foreach (ys, f)
6
7
8 val xs = List(11,21,31)
9 foreach (xs, println)
```




List Operation: Foreach

💡 Customize with changed function argument

```
1 def foreach (xs:List[Int], f:Int=>Unit) : Unit = xs match
2   case Nil    => ()
3   case y::ys =>
4     f (y)
5     foreach (ys, f)
6
7 def printY (y:Int) = println(s"y=$y")
8 val xs = List(11,21,31)
9 foreach (xs, printY)
```



List Operation: Foreach

💡 Generalize the type of list with parameters

```
1 def foreach [X] (xs:List[X], f:X=>Unit) : Unit = xs match
2   case Nil      => ()
3   case y::ys    =>
4     f (y)
5     foreach (ys, f)
6
7 def printLength (xs:List[Int]) = println (xs.length)
8 val xss = List(List(11,21,31),List(),List(41,51))
9 foreach (xss, printLength)
```



List Operation: Foreach

💡 Use a lambda expression (anonymous function)

```
1 def foreach [X] (xs:List[X], f:X=>Unit) : Unit = xs match
2   case Nil      => ()
3   case y::ys    =>
4     f (y)
5     foreach (ys, f)
6
7
8 val xss = List(List(11,21,31),List(),List(41,51))
9 foreach (xss, (xs:List[Int]) => println (xs.length))
```



List Operation: Foreach

Using the builtin `List` class `foreach` method

- Named method:

```
1 def print (x:Int) = println (x)
2 xs.foreach (print)
```

- Lambda expression

```
1 xs.foreach ((x:Int) => println (x))
```

- Types unnecessary if Scala can infer

```
1 xs.foreach (x => println (x))
```

- Anonymous intermediate function unnecessary

```
1 xs.foreach (println)
```



Types and Function Parameters

```
1 def foreach [X] (xs:List[X], f:X=>Unit) : Unit = ...
```

- `X` is a *type parameter*
 - Type parameters in square brackets
 - Value parameters in round brackets
 - Types before values
- `f` is a parameter of *function type*: `(X=>Unit)`
 - takes an argument of type `X`
 - returns a result of type `Unit`



Recursion

- Imperative programming typically favors
 - mutable data
 - iteration using loops (`while` , `for`)
- Functional programming typically favors
 - immutable data
 - iteration using recursion
- Recursion requires efficient method calls
- State of computation
 - Imperative: loop counters to access "global" mutable data
 - Recursion: arguments to recursive call



Exercise: Length of List

- Imperative implementation

```
1 def length (xs:List[Int]) : Int =  
2   var length: Int = 0  
3   var current = xs  
4   while current != Nil do  
5     length = length + 1  
6     current = current.tail  
7   end while  
8   length  
9 end length
```

- Recursive with pattern matching

```
1 def length [X] (xs: List[X]) : Int = xs match  
2   case Nil      => 0  
3   case _ :: ys => 1 + length (ys)  
4 end length
```



Iteration vs. Recursion

Example: `length (List (1, 2, 3))`

- Imperative iteration

```
--> current = 1::(2::(3::Nil)), length = 0
--> current = 2::(3::Nil),      length = 1
--> current = 3::Nil,           length = 2
--> current = Nil,              length = 3
```

- The state of the computation is in mutable variables

- Recursive iteration

```
--> length (1::(2::(3::Nil)))
--> 1 + length (2::(3::Nil))
--> 1 + (1 + length (3::Nil))
--> 1 + (1 + (1 + length (Nil)))
--> 1 + (1 + (1 + 0))
--> 1 + (1 + 1)
--> 1 + 2
--> 3
```

- The state of the computation is the expression



List Operation: Map

`foreach` visits every element

```
1 def foreach [X] (xs:List[X],  
2                 f:X=>Unit) : Unit =  
3   xs match  
4     case Nil    => ()  
5     case y::ys => f(y); foreach(ys, f)  
6 end foreach
```

- Result `Unit` : no result collected

`map` : visits and transforms every element

```
1 def map [X,Y] (xs:List[X],  
2               f:X=>Y) : List[Y] =  
3   xs match  
4     case Nil    => Nil  
5     case y::ys => f(y) :: map(ys, f)  
6 end map
```

- Result `List[Y]` : transformed copy

```
1 map(List(11,21,31), (y:Int) => "x=" + y)  
2 // res: List[String] = List("x=11","x=21","x=31")
```

- Builtin `xs.map((y:Int) => "x=" + y)`



Examples

- Increment each value in a list: `List[Int]=>List[Int]`

```
1 List(1, 2, 3, 4).map(x => x + 1)
2 // res: List[Int] = List(2, 3, 4, 5)
```

- Length of inner lists: `List[List[Int]]=>List[Int]`

```
1 List(List(11,21,31),Nil,List(41,51)).map(xs => xs.length)
2 // res: List[Int] = List(3, 0, 2)
```

- Length of strings: `List[String]=>List[Int]`

```
1 List("hi", "it's", "me").map(xs => xs.length)
2 // res: List[Int] = List(2, 4, 2)
```



List Operation: Filter

? Copy only elements that satisfy a predicate `f`

```
1 def filter [X] (xs:List[X], f:X=>Boolean) : List[X] = xs match
2   case Nil => Nil
3   case y::ys if f (y) => y :: filter (ys, f)
4   case _::ys => filter (ys, f)
5 end filter
```

```
1 val zs = (0 to 7).toList
2 filter(zs, (x => x % 3 != 0))
3 // res: List[Int] = List(1,2,4,5,7)
```



List and Set Comprehensions

Set Comprehensions List Comprehensions

$\{(m, n) \mid m \in \{0, \dots, 10\} \wedge n \in \{0, \dots, 10\} \wedge m \leq n\}$

- The set of all tuples (m, n) such that
 - m is in $\{0, \dots, 10\}$,
 - n is in $\{0, \dots, 10\}$,
 - the value of m is at most the value of n

- In many PLs
- SETL (1960s)
- Haskell `[ (m,n) | m <- [0..10], n <- [0..10], m <= n ]`
- Scala `for m <- 0 to 10; n <- 0 to 10; if m <= n yield (m,n)`
- Python `{(m, n) for m in range(0,11) for n in range(0,11) if m <= n}`
- JavaScript 1.7



For Comprehensions

Method **map**

```
1 xs.map (x => "x=" + x)
```

Method **foreach**

```
1 xs.foreach (x => println ("x=" + x))
```

For Comprehension **yield**

```
1 for x <- xs yield "x=" + x
```

For Comprehension **do**

```
1 for x <- xs do println ("x=" + x)
```



For Comprehensions

Method `filter`

```
1 zs.filter (z => z % 3 != 0)
```

Nested Methods

```
1 zs.filter (z => z % 3 != 0).  
2   map (z => "z=" + z)
```

For Comprehension

```
1 for z <- zs  
2   if z % 3 != 0 yield z
```

Combined For Comprehension

```
1 for z <- zs  
2   if z % 3 != 0 yield "z=" + z
```



For Comprehensions

- Multiple iterators

```
1 val xss = List( List(11,21,31), List(), List(41,51) )
2 for xs <- xss
3   x <- xs
4   yield (x, xs.length)
5 // res1: List[(Int, Int)] = List((11,3), (21,3), (31,3), (41,2), (51,2))
```

- Cross product of independent iterators

```
1 val xs = List(11,21,31)
2 val ys = List("a","b")
3 for x <- xs
4   y <- ys yield (x, y)
5 // res1: List[(Int, String)] = List(
6 //   (11,a), (11,b),
7 //   (21,a), (21,b),
8 //   (31,a), (31,b))
```

```
1 (for x <- (1 to 7)
2   y <- (1 to 9) yield (x, y)
3 ).length
4 // res1: Int = 63
```



For Comprehensions: Types

```
1 val xs = List(11, 21, 31)
2 val ys = List("a", "b")
3 for x <- xs
4     y <- ys yield (x, y)
```

- `xs : List[Int]`
- `ys : List[String]`
- Scala infers types for iterator variables
 - Inferred types

```
1 x : Int
2 y : String
```

- `yield` provides the type for the result

```
1 (x, y) : (Int, String)
2 for ... yield (x, y) : List[(Int, String)]
```




Summary

- Element-wise list processing is a universal, higher-order function

Higher-order functions

- Element-processing function is provided as argument
- `foreach` : visits every element
- `map` : creates transformed copy
- `filter` : creates filtered copy

List comprehensions

- Syntax mimics loops for expressing list comprehensions
- `for ... do` mimics `foreach`
- `for ... yield` mimics `map`
- `for ... if ... yield` mimics `filter (+ map)`



Recursion

- What does `f` do?

```
1 def f [X] (xs: List[X]) : List[X] = xs match
2   case Nil      => Nil
3   case y :: ys => f (ys) :: List (y)
```

- `f (Nil)`

- ```
1 f (Nil)
2 --> Nil
```

- `f (3::Nil)`

- ```
1 f (3::Nil)
2 --> f (Nil) ::: List (3)
3 --> Nil ::: List (3)
4 --> List (3)
```

- `f`

`(2::3::Nil)`

- ```
1 f (2::(3::Nil))
2 --> f (3::Nil) ::: List (2)
3 --> List (3) ::: List (2)
4 --> List (3, 2)
```

- `f`

`(1::2::3::Nil)`  
`)`

- ```
1 f (1::(2::(3::Nil)))
2 --> f (2::(3::Nil)) ::: List (1)
3 --> List (3, 2) ::: List (1)
4 --> List (3, 2, 1)
```

- Conclusion: `f` is `reverse`



Appending Lists

```
1 def append [X] (xs: List[X], ys: List[X]) : List[X] = xs match
2   case Nil      => ys
3   case z :: zs  => z :: append (zs, ys)
```

```
1 append (1::(2::Nil), 3::Nil)
2 --> 1::(append (2::Nil, 3::Nil))    // z = 1, zs = 2::Nil
3 --> 1::(2::(append (Nil, 3::Nil)))  // z = 2, zs = Nil
4 --> 1::(2::(3::Nil))                // z = 2, zs = Nil
```

- New elements `1` and `2` created
- List `3::Nil` is reused (shared)
- New list, but second part is shared!



Appending Lists

- `List` class has builtin method `:::`

```
1 scala> ((1 to 5).toList) ::: ((10 to 15).toList)
2 res1: List[Int] = List(1, 2, 3, 4, 5, 10, 11, 12, 13, 14, 15)
```



List Operation: Flatten

? Revisit the copy operation

```
1 def copy      [X] (xs:List[List[X]]) : List[List[X]] = xs match
2   case Nil     => Nil
3   case y::ys   => y :: copy      (ys)
4
5 val xss = List(List(11,21,31),List(),List(41,51))
6 copy(xss)
```

```
1 res1: List(List(11,21,31),List(),List(41,51))
```



List Operation: Flatten

💡 Create a copy with a flat structure: replace `::` with `:::`

```
1 def flatten [X] (xs:List[List[X]]) : List[X] = xs match
2   case Nil      => Nil
3   case y::ys    => y ::: flatten (ys)
4
5 val xss = List(List(11,21,31),List(),List(41,51))
6 flatten(xss)
```

```
1 res1: List(11,21,31,41,51)
```



List Operation: FlatMap

? Revisit method `map`

```
1 def map      [X,Y] (xs:List[X], f:X=>List[Y]) : List[List[Y]] = xs match
2   case Nil    => Nil
3   case y::ys => f(y) :: map      (ys, f)
4
5 val as = List(3,0,2)
6 map    (as, (x:Int) => (1 to x).toList)
```

```
1 res1: List(List(1,2,3), List(), List(1,2))
```



List Operation: FlatMap

💡 Create a transformed list with a flat structure: replace `::` with `:::`

```
1 def flatMap [X,Y] (xs:List[X], f:X=>List[Y]) : List[Y] = xs match
2   case Nil      => Nil
3   case y::ys    => f(y) ::: flatMap (ys, f)
4
5 val as = List(3,0,2)
6 flatMap(as, (x:Int) => (1 to x).toList)
```

```
1 res1: List(1,2,3,1,2):::Nil
```




List Operation: FlatMap

Argument and return types of `map` and `flatMap`

- `map: (List[X], X=>List[Y]) => List[List[Y]]`
 - Each element of result list is a `List[Y]`
 - Length of result = length of `xs`
- `flatMap: (List[X], X=>List[Y]) => List[Y]`
 - Each element of result list is a `Y`
 - Length of result = sum of lengths of each `List[Y]`



For Comprehensions

Method `flatMap`

```
1 xss.flatMap (x=>x) // same as xss.flatten
```

For Comprehension

```
1 for xs <- xss; x <- xs yield x
```