

CSC 347 - Concepts of Programming Languages

Collections Processing: Fold

Instructor: Stefan Mitsch



Learning Objectives

- ❓ How to combine collection elements into an aggregate result?
 - Express result aggregation using `fold`
 - Identify the difference between `foldLeft` and `foldRight`



Exercise: Sum the Elements of a List

? Express by iterating through the list

Java

```
1 int sum (List<Int> xs) {  
2     int result = 0;  
3     for (x : xs) result += x;  
4     return result;  
5 }
```

Scala

```
1 def sum (xs:List[Int]) : Int = {  
2     var result = 0  
3     for x <- xs do result += x  
4     result  
5 } ensuring { case result => ??? }
```

- Want a concise function to aggregate all elements of `xs` into a single result



Exercise: Sum the Elements of a List

? Express with a recursive function

```
1 def sum (xs:List[Int])          : Int = xs match
2   case Nil      => 0
3   case y :: ys => y + sum (ys)
4
5 val xs = List(11,21,31)
6 sum (xs)
```

```
1 sum(11::21::31::Nil)
2 --> sum(11::21::31::Nil)
3 --> 11 + sum(21::31::Nil)
4 --> 11 + (21 + sum(31::Nil))
5 --> 11 + (21 + (31 + sum(Nil)))
6 --> 11 + (21 + (31 + 0))
7 --> 11 + (21 + 31)
8 --> 11 + 52
9 --> 63 = (11 + (21 + (31 + 0)))
```



Exercise: Sum the Elements of a List

? With a different zero element

```
1 def sum (xs:List[Int], z:Int = 0) : Int = xs match
2   case Nil      => z
3   case y :: ys => y + sum (ys, z)
4
5 val xs = List(11,21,31)
6 sum (xs)
```

```
1 sum(11::21::31::Nil)
2 --> sum(11::21::31::Nil, 0)
3 --> 11 + sum(21::31::Nil, 0)
4 --> 11 + (21 + sum(31::Nil, 0))
5 --> 11 + (21 + (31 + sum(Nil, 0)))
6 --> 11 + (21 + (31 + 0))
7 --> 11 + (21 + 31)
8 --> 11 + 52
9 --> 63 = (11 + (21 + (31 + 0)))
```



Exercise: Sum the Elements of a List

? Sum of elements in a list computing forward

```
1 def sum (xs:List[Int], z:Int = 0) : Int = xs match
2   case Nil      => z
3   case y :: ys => sum (ys, z + y)
4
5 val xs = List(11,21,31)
6 sum (xs)
```

```
1 sum(11::21::31::Nil)
2 --> sum(11::21::31::Nil, 0)
3 --> sum(21::31::Nil, 11)
4 --> sum(31::Nil, 32)
5 --> sum(Nil, 63)
6 -->
7 -->
8 -->
9 --> 63 = (((0 + 11) + 21) + 31)
```



Folds

💡 generalize the `+` operation

```
1 def sum          (xs:List[Int], z:Int)          : Int =  
2   xs match  
3     case Nil      => z  
4     case y::ys    => sum          (ys, z + y)  
5  
6 val xs = List(11,21,31)  
7 sum    (xs, 0)
```

```
1 res1: Int = 63
```



Folds

💡 generalize the `+` operation

```
1 def foldLeft (xs:List[Int], z:Int, f:((Int,Int)=>Int)) : Int =  
2   xs match  
3     case Nil      => z  
4     case y::ys    => foldLeft (ys, f(z,y), f)  
5  
6 val xs = List(11,21,31)  
7 foldLeft (xs, 0, _+_)
```

```
1 res1: Int = 63
```




Folds

? Change the return type

```
1 def foldLeft (xs:List[Int], z:String, f:(String,Int)=>String) : String =  
2   xs match  
3     case Nil      => z  
4     case y::ys    => foldLeft (ys, f(z, y), f)  
5  
6 val xs = List(11,21,31)  
7 foldLeft (xs, "", _ + " " + _)
```

```
1 res1: String = "11 21 31 "
```



Folds

? Changing the parameter type

```
1 def foldLeft (xs:List[List[Int]], z:Int, f:(Int,List[Int])=>Int) : Int =  
2   xs match  
3     case Nil    => z  
4     case y::ys => foldLeft (ys, f(z, y), f)  
5  
6 val xss = List(List(11,21,31),List(),List(41,51))  
7 foldLeft (xss, 0, _ + _.length)
```

```
1 res1: Int = 5
```



Folds

💡 Abstracting the type

```
1 def foldLeft [Z,X] (xs:List[X], z:Z, f:((Z,X)=>Z)) : Z =  
2   xs match  
3     case Nil      => z  
4     case y::ys    => foldLeft (ys, f(z,y), f)  
5  
6 val xs = List(11,21,31)  
7 foldLeft (xs, "!", (z:String,x:Int) => z + " " + x)
```

```
1 res1: String = "! 11 21 31"
```



Fold Left vs. Fold Right

Fold Left

```
1 def foldLeft [Z,X] (xs:List[X], z:Z, f:((Z,X)=>Z)) : Z =  
2   xs match  
3     case Nil    => z  
4     case y::ys => foldLeft (ys, f(z,y), f)  
5  
6 val xs = List(11,21,31)  
7 foldLeft (xs, "!", (z:String,x:Int) => z + " " + x)
```

```
1 res1: String = "! 11 21 31"
```

Fold Right

```
1 def foldRight [X,Z] (xs:List[X], z:Z, f:((X,Z)=>Z)) : Z =  
2   xs match  
3     case Nil    => z  
4     case y::ys => f (y, foldRight (ys, z, f))  
5  
6 val xs = List(11,21,31)  
7 foldRight (xs, "!", (x:Int,z:String) => x + " " + z)
```

```
1 res1: String = "11 21 31 !"
```



Folds Builtin in Lists

- Scala `List` class has `fold` methods (curried!)

```
1 xss.foldLeft (0) ((z,xs)=>z + xs.length)
```



Fold Left vs. Fold Right

```
1 def foldLeft [Z,X] (xs:List[X], z:Z, f:((Z,X)=>Z)) : Z = xs match {  
2   case Nil => z  
3   case y::ys => foldLeft (ys, f(z,y), f)  
4 }
```

```
1 def foldRight [X,Z] (xs:List[X], z:Z, f:((X,Z)=>Z)) : Z = xs match {  
2   case Nil => z  
3   case y::ys => f (y, foldRight (ys, z, f))  
4 }
```

- `foldLeft` is *tail recursive*: `return foldLeft (ys, f(z, y))`
 - apply `f` to the head and the accumulated result
 - recursive call on the tail
 - base case used with first element
- `foldRight` is *recursive into an argument*:
 - `return f (y, foldRight (ys, z))`
 - recursive call on the tail
 - apply `f` to the head and result of recursion
 - base case used with last element

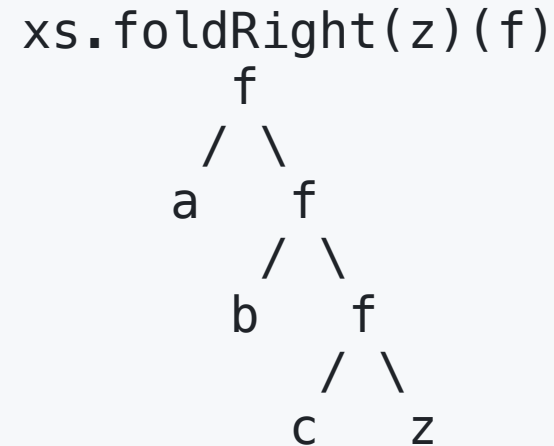
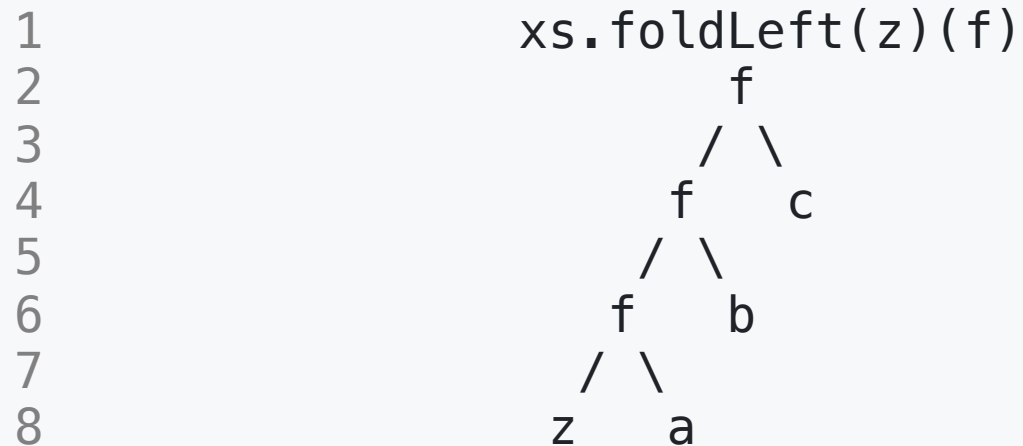


Fold Left vs. Fold Right

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

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

```
1 val xs = List(a, b, c)
2 foldLeft (xs, z, f) == f( f( f(z,a),b),c)
3 foldRight(xs, z, f) == f(a, f(b, f(c,z)))
```





Folds are Universal

```
1 def sum      (xs: List[Int])      = xs.foldLeft(0)(_+_)  
2 def prod     (xs: List[Int])      = xs.foldLeft(1)(_*_)  
3 def or       (xs: List[Boolean])  = xs.foldLeft(false)(_||_)  
4 def and      (xs: List[Boolean])  = xs.foldLeft(true)(_&&_)  
5 def append   [X] (xs: List[X])(ys: List[X]) = xs.foldRight(ys)(_::_)  
6 def flatten  [X] (xs: List[List[X]]) = xs.foldLeft(List[X])(_:::_)  
7 def length   [X] (xs: List[X])    = xs.foldLeft(0)((z,x)=>z+1)  
8 def reverse  [X] (xs: List[X])    = xs.foldRight(List[X])(_:::_)  
9 def map      [X,Y] (xs: List[X], f: X=>Y) = xs.foldRight(List[Y])(f(_)::_)  
10 def filter  [X] (xs: List[X], f: X=>Boolean) = xs.foldRight(List[X])(_:::_ if f(_) else _)
```

- Lots of [examples](#)
- Tutorial on [universality of folds](#)



Summary

- Folds are universal functions to combine list elements into an aggregate result
- `foldRight` folds from the right (zero element combined with last element)
- `foldLeft` folds from the left (zero element combined with list head)