

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 x::rest => x + sum (rest)
4
5 val xs = List(11,21,31)
6 sum (xs)
```

```
sum(11::21::31::Nil)
--> sum(11::21::31::Nil)
--> 11 + sum(21::31::Nil)
--> 11 + (21 + sum(31::Nil))
--> 11 + (21 + (31 + sum(Nil)))
--> 11 + (21 + (31 + 0))
--> 11 + (21 + 31)
--> 11 + 52
--> 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 x::rest => x + sum (rest, z)
4
5 val xs = List(11,21,31)
6 sum (xs)
```

```
sum(11::21::31::Nil)
--> sum(11::21::31::Nil, 0)
--> 11 + sum(21::31::Nil, 0)
--> 11 + (21 + sum(31::Nil, 0))
--> 11 + (21 + (31 + sum(Nil, 0)))
--> 11 + (21 + (31 + 0))
--> 11 + (21 + 31)
--> 11 + 52
--> 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 x::rest => sum (rest, z + x)
4
5 val xs = List(11,21,31)
6 sum (xs)
```

```
sum(11::21::31::Nil)
--> sum(11::21::31::Nil, 0)
--> sum(21::31::Nil, 11)
--> sum(31::Nil, 32)
--> sum(Nil, 63)
-->
-->
-->
--> 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 x::rest => sum          (rest, z + x)  
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 x::rest => foldLeft (rest, f(z,x), 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 x::rest => foldLeft (rest, f(z, x), 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:String, f:(String,List[Int])=>String) : String =  
2   xs match  
3     case Nil      => z  
4     case x::rest => foldLeft (rest, f(z, x), f)  
5  
6 val xss = List( List(11,21,31), List(), List(41,51) )  
7 foldLeft (xss, "[ ", _ + " " + _.length) + "]"
```

```
1 res1: Int = "[ 3 0 2 ]"
```



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 x::rest => foldLeft (rest, f(z,x), 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 x::rest => foldLeft (rest, f(z,x), 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 x::rest => f (x, foldRight (rest, 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)
```

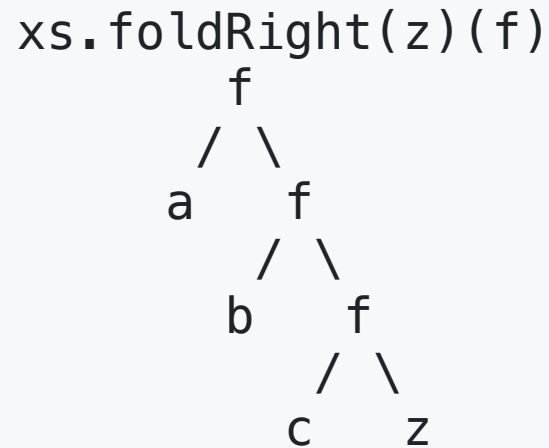
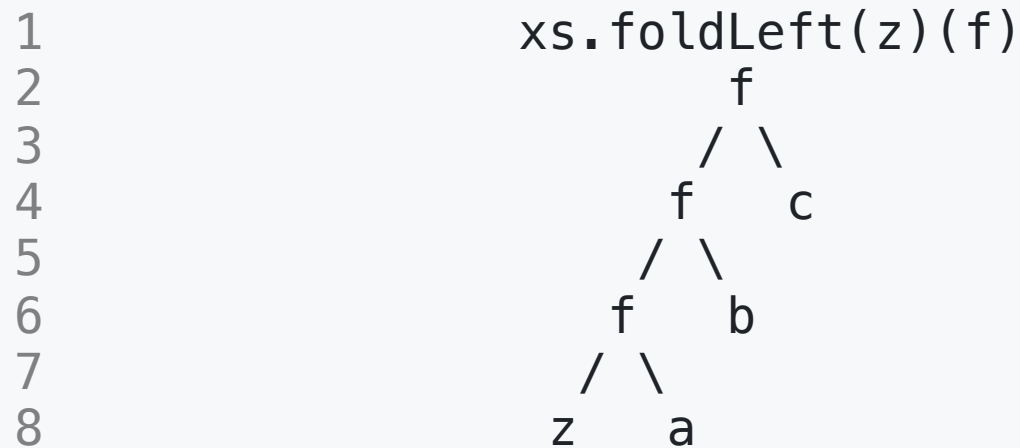


Fold Left vs. Fold Right

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

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





Fold vs. Reduce

Fold

- Starts from a user-provide initial value
- Result type may differ from collection element type

```
1 List(1,2,3).fold(0)(_ + _)
```

- Change result type in a fold, provide initial element of result type

```
1 List(1,2,3).fold("!")(_.toString + _)
```

Reduce

- Starts from first element of collection
- Result type same as collection element type

```
1 List(1,2,3).reduce(_+_)
```

- Change result type with map before reduce, add initial element

```
1 "!" + List(1,2,3).map(_.toString).reduce(_+_)
```



Exceptions when Folding

- Example: turn list of tuples into sum of divisions

```
1 List(4->2,9->3).foldLeft(0) { case (acc,(n,d)) => acc + n/d }
```

- Division by 0

```
1 List(4->2,5->0,9->3).foldLeft(0) { case (acc,(n,d)) => acc + n/d }
```

- Collect first exception in Either

```
1 List(4->2,5->0,9->3).foldLeft[Either[Throwable,Int]](Right(0)) {  
2     case (acc,(n,d)) => acc.flatMap {  
3         a => try  
4             Right(a + n/d)  
5             catch  
6                 e => Left(e)  
7     }  
8 }
```



Folds are Universal

```
1 def sum          (xs: List[Int])
2 def prod         (xs: List[Int])
3 def or           (xs: List[Boolean])
4 def and          (xs: List[Boolean])
5 def append [X] (xs: List[X])(ys: List[X])
6 def flatten [X] (xs: List[List[X]])
7 def length [X] (xs: List[X])
8 def reverse [X] (xs: List[X])
9 def map      [X,Y] (xs: List[X], f: X=>Y)
10 def filter  [X] (xs: List[X], f: X=>Boolean)
11 def reduce  [X] (xs: List[X], f: (X,X)=>X)
```



Summary

- Folds are universal functions to combine list elements into an aggregate result

Fold from the left **foldLeft**

- Zero element combined with head

```
1 xs.foldLeft(z)(f)
2
3      f
4     / \
5    f   c
6   / \
7  f   b
8 /  \
z    a
```

Fold from the right **foldRight**

- Zero element combined with last

```
1 xs.foldRight(z)(f)
2
3      f
4     / \
5    a   f
6       / \
7      b   f
8         / \
          c   z
```

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



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])((x,zs)=>zs::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])((x,zs)=>if f(x) then x::zs else zs)
```