

Matt Malone's Old-Fashioned Software Development Blog

July 30, 2009

Lots And Lots Of foldLeft Examples

Posted by Matt under [General](#), [Scala](#) | Tags: [example](#), [examples](#), [fold](#), [foldLeft](#), [foldRight](#), [Scala](#) | [\[26\] Comments](#)

In [my last post](#) I reviewed the implementation of `scala.List`'s `foldLeft` and `foldRight` methods. That post included a couple of simple examples, but today I'd like to give you a whole lot more. The `foldLeft` method is extremely versatile. It can do thousands of jobs. Of course, it's not the best tool for EVERY job, but when working on a list problem it's a good idea to stop and think, "Should I be using `foldLeft`?"

Below, I'll present a list of problem descriptions and solutions. I thought about listing all the problems first, and then the solutions, so the reader could work on his own solution and then scroll down to compare. But this would be very annoying for those who refuse, against my strenuous urging, to start up a Scala interpreter and try to write their own solution to each problem before reading my solution.

Sum

Write a function called 'sum' which takes a `List[Int]` and returns the sum of the Ints in the list. Don't forget to use `foldLeft`.

```
1 | def sum(list: List[Int]): Int = list.foldLeft(0)((r,c) => r+c)
2 | def sum(list: List[Int]): Int = list.foldLeft(0)(_+_)
```

I'll explain this first example in a bit more depth than the others, just to make sure we all know how `foldLeft` works.

These two definitions above are equivalent. Let's examine the first one. The `foldLeft` method is called on the list parameter. The first parameter is 0. This is the starting value, and the value that will be returned if list is empty. The second parameter is a function literal. It takes parameters 'r' (for result) and 'c' (for current) and returns the sum of these two values. Scala is smart enough to figure out that since the first parameter (0) is an Int, the 'r' parameter must also be an Int. The initial value is always the same type as 'r'. Scala can also tell that since 'list' is a `List[Int]` the 'c' parameter must also be an Int, so we don't have to specify their types in the parameter list.

The foldLeft method takes that initial value, 0, and the function literal, and it begins to apply the function on each member of the list (parameter 'c'), updating the result value (parameter 'r') each time. That result value that we call 'r' is sometimes called the accumulator, since it accumulates the results of the function calls.

In the first definition, foldLeft's second parameter (a function literal) uses explicitly named parameters. Notice that 'r' and 'c' are each referred to exactly once in the function literal, and in the same order as the parameter list. When function literal parameters are used in this way (once each, same order) you can use the shorthand demonstrated in the second definition. The first '_' stands for 'r', and the second one stands for 'c'.

Product

Now that you've got the idea, try this one. Write a function that takes a List[Int], and returns the product (of multiplication) of all the Ints in the list. It will be similar to the 'sum' function, but with a couple of differences.

```
1 | def product(list: List[Int]): Int = list.foldLeft(1)(_*_)
```

Did you get it? It's the same as 'sum' with two exceptions. The initial value is now 1, and the function literal's parameters are multiplied instead of added. If the initial value were 0, as in 'sum', then the function would always return 0.

Count

This one's a little different. Write a function that takes a List[Any] and returns the number of items in the list. Don't just call list.length()! Implement it using foldLeft.

```
1 | def count(list: List[Any]): Int =  
2 |   list.foldLeft(0)((sum,_) => sum + 1)
```

First, we pick our initial value. Remember that this is the value that will be returned for an empty list. An empty list has 0 elements, so we use 0. What function do we want to apply for every item in the list? We just want to increase the result value by one. We call that parameter 'sum' in this solution. We don't care about the actual value of each list element, so we call the second parameter '_', which means it should be discarded.

Average

Here's a fun one. Write a function that takes a `List[Double]` and returns the average of the list's values. There are two ways to go about this one. You could combine two of the previous solutions, using two `foldLeft` calls, or you could combine them into a single `foldLeft`. Try to find both solutions.

```

1 | def average(list: List[Double]): Double =
2 |     list.foldLeft(0.0)(_+_ ) / list.foldLeft(0.0)((r,c) => r+1)
3 |
4 | def average(list: List[Double]): Double = list match {
5 |     case head :: tail => tail.foldLeft( (head,1.0) )((r,c) =>
6 |         ((r._1 + (c/r._2)) * r._2 / (r._2+1), r._2+1) )._1
7 |     case Nil => NaN
8 | }
```

The first solution is pretty easy and combines the 'sum' and 'count' solutions. In real life, of course, you wouldn't use `foldLeft` to find the length of the list. You'd just use the `length()` method. Other than that, though, this is a perfectly sensible solution.

The second solution is more complex. First, the list is matched against two patterns. It is either interpreted as a head item followed by a tail, or as an empty list (`Nil`). If it's empty, the function returns the same thing as the first solution, `NaN` (Not a Number) because you can't divide by 0.

If the list is not empty, we use a `Pair` as our initial value. A `Pair` is just an ordered pair of values. It's a convenient way to bundle values together. We use it when we need to keep track of more than one accumulator value. In this case, we want to keep track of the average "so far" and also the number of values that the average represents. If the function literal were just passed the average so far, it wouldn't know how to weight the next value. Members of a `Pair` are accessed using special methods called `'_1'` and `'_2'`. You can have groupings longer than 2, also. These are named `Tuple3`, `Tuple4`, and so on. In fact, `Pair` is just an alias of `Tuple2`. Notice that we didn't use the word `Pair` or `Tuple2` anywhere in the code. If you enclose a comma-delimited series of values in parentheses, Scala converts that series into the appropriate `TupleX`.

After we have built up the result, it is a `Pair` containing the average and the number of items in the list. We only want to return the average so we call `'_1'` on the result of `foldLeft`.

Last

Whew! That one was a little tough. Here's an easier one. Given a `List[A]` return the last value in the list. Again, no using `List`'s `last()` method.

```

1 | def last[A](list: List[A]): A =
3 of 12 |     list.foldLeft[A](list.head)((_, c) => c)
```

Easy! Mostly. You'll notice that we're using a type parameter, *A*, in this one. If you're not familiar with type parameters, too bad. I can't explain them here. Suffice it to say that our use of *A* here allows us to take a list of any type of contents, and return a result of just that type. So Scala knows that when this is called on a `List[Int]`, it will return an `Int`. When it's called on a `List[String]`, it returns a `String`.

First, we pick an initial value. For the empty list the concept of a last item doesn't make any sense, so forget that. We can use any value, so long as it's of type *A*. `list.head` is convenient, so that's our initial value. The function literal is the simplest we've seen. For each item in the list, it just returns that item itself. So when it gets to the end of the list, the accumulator holds the last item. We don't use the accumulator value in the function literal, so it gets parameter name `'_'`.

Penultimate

Write a function called 'penultimate' that takes a `List[A]` and returns the penultimate item (i.e. the next to last item) in the list. Hint: Use a tuple.

```
1 | def penultimate[A](list: List[A]): A =  
2 |   list.foldLeft( (list.head, list.tail.head) )((r, c) => (r._2, c) )._1
```

This one is very much like the function 'last', but instead of keeping just the current item it keeps a `Pair` containing the previous and current items. When `foldLeft` completes, its result is a `Pair` containing the next-to-last and last items. The `"_1"` method returns just the penultimate item.

Contains

Write a function called 'contains' that takes a `List[A]` and an item of type *A*, and returns true if the item is one of the members of the list, and false if it isn't.

```
1 | def contains[A](list: List[A], item: A): Boolean =  
2 |   list.foldLeft(false)(_ || _==item)
```

We choose an initial value of false. That is, we'll assume the item is not in the list until we can prove otherwise. We use each of the two parameters exactly once and in the proper order, so we can use the `'_'` shorthand in our function literal. That function literal returns the result so far (a `Boolean`) `ORed` with a comparison of the current item and the target value. If the target is ever found, the accumulator becomes true and stays true as `foldLeft` continues.

Get

Write a function called 'get' that takes a List[A] and an index Int, and returns the list value at the index position. Throw an exception if the index is out of bounds.

```

1 | def get[A](list: List[A], idx: Int): A =
2 |   list.tail.foldLeft((list.head,0)) {
3 |     (r,c) => if (r._2 == idx) r else (c,r._2+1)
4 |   } match {
5 |     case (result, index) if (idx == index) => result
6 |     case _ => throw new Exception("Bad index")
7 |   }

```

This one has two parts. First there's the foldLeft, and the result is pattern matched. The foldLeft is pretty easy to follow. The accumulator is a Pair containing the current item and the current index. The current item keeps updating and the current index keeps incrementing until the current index equals the passed in idx. Once the correct index is found the same accumulator is returned over and over. This works fine if idx parameter is in bounds. If it's out of bounds, though, the foldLeft just returns a Pair containing the last item and the last index. That's where the pattern match comes in. If the Pair contains the right index then we use the result item. Otherwise, we throw an exception.

MimicToString

Write a function called 'mimicToString' that mimics List's own toString method. That is, it should return a String containing a comma-delimited series of string representations of the list contents with "List(" on the left and ")" on the right.

```

1 | def mimicToString[A](list: List[A]): String = list match {
2 |   case head :: tail => tail.foldLeft("List(" + head)(_ + ", " + _) + ")")
3 |   case Nil => "List()"
4 | }

```

This one also uses a pattern match, but this time the match happens first. The pattern match just treats the empty list as a special case. For the general case (a non-empty list) we use, of course, foldLeft. The accumulator starts out as "List(" + the head item. Then each remaining item (notice foldLeft is called on tail) is appended with a leading ", " and a final ")" is added to the result of foldLeft.

Reverse

This one's kind of fun. Make sure to try it before you look at my solution. Write a function called 'reverse' that takes a List and returns the same list in reverse order.

```

1 | def reverse[A](list: List[A]): List[A] =
2 |   list.foldLeft(List[A]())(r,c => c :: r)

```

instead spell out the List type so that Scala will know what type to make 'r'. As I say, we start with the empty list which is sensible because the reverse of an empty list is an empty list. Then, as we go through the list, we place each item at the front of the accumulator. So the item at the front of list becomes the last item in the accumulator. This goes on until we reach the end of list, and that last member of list goes onto the front of the accumulator. It's a really neat and tidy solution.

Unique

Write a function called 'unique' that takes a List and returns the same List, but with duplicated items removed.

```
1 | def unique[A](list: List[A]): List[A] =  
2 |   list.foldLeft(List[A]()) { (r,c) =>  
3 |     if (r.contains(c)) r else c :: r  
4 |   }.reverse
```

As usual, we start with an empty list. foldLeft looks at each list item and if it's already contained in the accumulator then it stays as it is. If it's not in the accumulator then it's appended. This code bears a striking similarity to the 'reverse' function we wrote earlier except for the "if (r.contains(c)) r" part. Because of this, the foldLeft result is actually the original list with duplicates removed, but in **reverse** order. To keep the output in the same order as the input, we add the call to reverse. We could also have chained on the foldLeft from the 'reverse' function, like so:

```
1 | def unique[A](list: List[A]): List[A] =  
2 |   list.foldLeft(List[A]()) { (r,c) =>  
3 |     if (r.contains(c)) r else c :: r  
4 |   }.foldLeft(List[A]())((r,c) => c :: r)
```

ToSet

Write a function called 'toSet' that takes a List and returns a Set containing the unique elements of the list.

```
1 | def toSet[A](list: List[A]): Set[A] =  
2 |   list.foldLeft(Set[A]())( (r,c) => r + c)
```

Super easy one. You just start out with an empty Set, which would be the right answer for an empty List. Then you just add each list item to the accumulator. Since the accumulator is a Set, it takes care of eliminating duplicates for you.

Double

Write a function called 'double' that takes a List and a new List in which each item appears twice in a row. For example double(List(1, 2, 3)) should return List(1, 1, 2, 2, 3, 3).

```
1 | def double[A](list: List[A]): List[A] =
2 |   list.foldLeft(List[A]())(r,c => c :: c :: r).reverse
```

Again, pretty easy. Are you starting to see a pattern. When you use foldLeft to transform one list into another, you usually end up with the reverse of what you really want.

Alternately, you could have used the foldRight method instead. This does the same thing as foldLeft, except it accumulates its result from back to front instead of front to back. I can't recommend using it, though, due to problems I point out in [my other post on foldLeft and foldRight](#). But here's what it would look like:

```
1 | def double[A](list: List[A]): List[A] =
2 |   list.foldRight(List[A]())(c,r => c :: c :: r)
```

InsertionSort

This one takes some thinking. Write a function called 'insertionSort' that uses foldLeft to sort the input List using the insertion sort algorithm. Try it on your own before you look at the solution.

Need a hint? Use List's 'span' method.

Did you find a solution? Here's mine:

```
1 | def insertionSort[A <% Ordered[A]](list: List[A]): List[A] =
2 |   list.foldLeft(List[A]())(r,c =>
3 |     val (front, back) = r.span(_ < c)
4 |     front ::: c :: back
5 |   }
```

First, the type parameter ensures that we have elements that can be arranged in order. We start, predictably, with an empty list as our initial accumulator. Then, for each item we assume the accumulator is in order (which it always will be), and use span to split it into two sub-lists: all already-sorted items less than the current item, and all already-sorted items greater than or equal to the current item. We put the current item in between these two and the accumulator remains sorted. This is, of course, not the fastest way to sort a list. But it's a neat foldLeft trick.

Pivot

Speaking of sorting, you can implement *part* of quicksort with foldLeft, the pivot. Write a function called 'pivot' that takes a List, and returns a Tuple3 containing: (1) a list of all elements less than the original list's first element, (2) the first element, and (3) a List of all elements greater than or equal to the first element.

```
1 | def pivot[A <% Ordered[A]](list: List[A]): (List[A],A,List[A]) =
2 |   list.tail.foldLeft[(List[A],A,List[A])]( (Nil, list.head, Nil) ) {
3 |     (result, item) =>
4 |     val (r1, pivot, r2) = result
5 |     if (item < pivot) (item :: r1, pivot, r2) else (r1, pivot, item :: r2)
6 |   }
```

We're using the first element, head, as the pivot value, so we skip the head and call foldLeft on list.tail. We initialize the accumulator to a Tuple3 containing the head element with an empty list on either side. Then for each item in the list we just pick which of the two lists to add to based on a comparison with the pivot value.

If you take the additional step of turning this into a recursive call, you can implement a quicksort algorithm. It probably won't be a very efficient one because it will involve a lot of building and rebuilding lists. Give it a try if you like, and then look at my solution:

```
1 | def quicksort[A <% Ordered[A]](list: List[A]): List[A] = list match {
2 |   case head :: _ :: _ =>
3 |     println(list)
4 |     list.foldLeft[(List[A],List[A],List[A])]( (Nil, Nil, Nil) ) {
5 |       (result, item) =>
6 |       val (r1, r2, r3) = result
7 |       if (item < head) (item :: r1, r2, r3)
8 |       else if (item > head) (r1, r2, item :: r3)
9 |       else (r1, item :: r2, r3)
10 |    } match {
11 |      case (list1, list2, list3) =>
12 |        quicksort(list1) ::: list2 ::: quicksort(list3)
13 |    }
14 |   case _ => list
15 | }
```

Basically, for all lists that have more than 1 element the function chooses the head element as the pivot value, uses foldLeft to divide the list into three (less than, equal to, and greater than the pivot), recursively sorts the less-than and greater-than lists, and knits the three together.

Encode

Ok, we got a little into the weeds with that last one. Here's a simpler one. Write a function called 'encode' that takes a List and returns a list of Pairs containing the original values and the number of times they are repeated. So passing List(1, 2, 2, 2, 2, 2, 3, 2, 2) to encode will return List((1, 1), (2, 5), (3, 1), (2, 2)).

```
1 | def encode[A](list: List[A]): List[(A, Int)] =
2 |   list.foldLeft(List[(A, Int)]()) { (r, c) =>
3 |     r match {
4 |       case (value, count) :: tail =>
5 |         if (value == c) (c, count+1) :: tail
6 |         else           (c, 1) :: r
7 |       case Nil =>
8 |         (c, 1) :: r
9 |     }
10 |   }.reverse
```

Decode

You knew this was coming. Write a function called 'decode' that does the opposite of encode. Calling 'decode(encode(list))' should return the original list.

```
1 | def decode[A](list: List[(A, Int)]): List[A] =
2 |   list.foldLeft(List[A]() { (r, c) =>
3 |     var result = r
4 |     for (_ <- 1 to c._2) result = c._1 :: result
5 |     result
6 |   }).reverse
```

Encode and decode could both have been written by using foldRight and dropping the call to reverse.

Group

One last example. Write a function called 'group' that takes a List and an Int size that groups elements into sublists of the specified sizes. So calling "group(List(1, 2, 3, 4, 5, 6, 7), 3)" should return List(List(1, 2, 3), List(4, 5, 6), List(7)). Don't forget to make sure list items are in the right order. Try it yourself before you look at the solution below.

```



1 | def group[A](list: List[A], size: Int): List[List[A]] =
2 |   list.foldLeft( (List[List[A]](), 0) ) { (r,c) => r match {
3 |     case (head :: tail, num) =>
4 |       if (num < size) ( (c :: head) :: tail , num + 1 )
5 |       else           ( List(c) :: head :: tail , 1 )
6 |     case (Nil, num) => (List(List(c)), 1)
7 |   }
8 | }. _1.foldLeft(List[List[A]]())( (r,c) => c.reverse :: r)

```

This code uses the first foldLeft to group the items in a way that's convenient to list operations, and that last foldLeft to fix the order, which would otherwise be wrong in both the outer and inner lists.

The End!

That's all for now. If you know of any neat foldLeft tricks, please do leave a comment. I'd be interested to hear about it.

Don't forget to  [subscribe to my RSS feed](#), or  [follow this blog on Twitter](#).

Copyright © 2009 Matthew Jason Malone



26 Responses to “Lots And Lots Of foldLeft Examples”

1. Sekib Says:

August 6, 2009 at 4:02 pm

Hi Matthew,

thanks for the articles. I really enjoyed reading 'em all.

Here are some examples on learning scala with your posts (MyList extension):

```
def foldLeft[B](z: B)(f: (B, A) => B): B = {
  def loop(l: MyList[A], acc: B, ff: (B, A) => B): B = {
    if (!l.isEmpty) loop(l.tail, ff(acc, l.head), ff)
    else acc
  }

  loop(this, z, f)
}
```

```
def foldRight[B](z: B)(f: (A, B) => B): B =
  this.reverse.foldLeft(z)((res, cur) => f(cur, res))
```

```
def foreach(f: A => Unit) {
  this.foldLeft()((res, cur) => f(cur))
}
```

```
def forall(f: A => Boolean): Boolean =
  this.foldLeft[Boolean](true)((res, cur) => res && f(cur))
```

```
def length(): Int =
  this.foldLeft(0)((sum, _) => sum + 1)
```

```
def map[B](f: A => B): MyList[B] =
  this.foldRight[MyList[B]](MyList[B]())((cur, res) => f(cur) :: res)
  // this.foldLeft[MyList[B]](MyList[B]())((res, cur) => f(cur) :: res).reverse
```

```
def filter(f: A => Boolean): MyList[A] =
  this.foldLeft[MyList[A]](MyList())((res, cur) => if (f(cur)) {cur :: res} else res).reverse
```

```
def reduceLeft[B >: A](f: (B, A) => B): B =
  if (isEmpty) throw new UnsupportedOperationException("empty.reduceLeft")
  else tail.foldLeft[B](head)(f)
```

```
def reduceRight[B >: A](f: (A, B) => B): B =
  if (isEmpty) throw new UnsupportedOperationException("empty.reduceLeft")
  else tail.foldRight[B](head)(f)
```

```
def exists(p: A => Boolean): Boolean =
  this.foldLeft[Boolean](false)((res, cur) => res || p(cur))
```

```
// no foldLeft but functional and tail-recursive
def indices: MyList[Int] = {
  def loop(l: MyList[A], res: MyList[Int], idx: Int): MyList[Int] =
    if (!l.isEmpty) loop(l.tail, res, idx+1)
    else res

  loop(this, MyList[Int](), 0).reverse
}
```

```
def dropWhile(p: A => Boolean): MyList[A] = {
  def loop(l: MyList[A], f: A => Boolean): MyList[A] =
```

```
if (!l.isEmpty && f(l.head)) loop(l.tail, f)
else l
```

```
loop(this, p)
}
```

```
def takeWhile(p: A => Boolean): MyList[A] = {
def loop(l: MyList[A], res: MyList[A], f: A => Boolean): MyList[A] =
if (!l.isEmpty && f(l.head)) loop(l.tail, l.head :: res, f)
else res
```

```
loop(this, MyList[A](), p) reverse
}
```

```
def drop(n: Int): MyList[A] = {
def loop(l: MyList[A], idx: Int, cur: Int): MyList[A] =
if (!l.isEmpty && cur < n) loop(l.tail, idx, cur+1)
else l
```

```
loop(this, n, 0)
}
```

```
def reverseMap[B](f: A => B): MyList[B] = {
def loop(l: MyList[A], res: MyList[B]): MyList[B] = {
if(l.isEmpty) res
else loop(l.tail, f(l.head) :: res)
}
loop(this, MyList())
}
```

Sorry if the formatting is broken :)

Reply

1. Matt Says:

August 6, 2009 at 6:42 pm

Glad to hear that it has been useful for you. Implementing the list operations yourself is a great way to learn how they work and to get used to the functional style of programming. Thanks a lot for the contribution, Sekib.

Reply

2. Scala Code Review: foldLeft and foldRight « Matt Malone's Old-Fashioned Software Development Blog Says:

August 20, 2009 at 11:13 pm

[...] I've created a new post in which I list lots and lots of foldLeft examples in case you'd like to learn more about what folding can [...]

Reply

3. Walter Says:

December 21, 2009 at 1:09 pm

You showed some very nice examples, but only for basic operations of built-in types. I'm missing some real-word examples with Domain objects.

Reply

1. Matt Says:

December 22, 2009 at 10:30 pm

The same principles apply for domain objects. You build up your accumulator by applying some code for each item in the list. For example, say you had a class called Vector2D. It stores a 2-dimensional vector using Cartesian coordinates. Like so:

```
class Vector2D(val x: Double, val y: Double) {  
  def +(that: Vector2D) = new Vector2D(this.x + that.x, this.y + that.y)  
}
```

Now, if you have several vectors that you want to sum, you do it just as for integers.

```
val finalVector = vectorList.foldLeft(new Vector(0,0))(_+_)
```

Or say you had a list of employees and you want to calculate the yearly payroll:

```
val payroll = employeeList.foldLeft(0)(_+_salary)
```

Or perhaps a user has selected a number of Products and he is ready to purchase:

```
val purchaseOrder = productList.foldLeft(new PurchaseOrder)(_addProduct(_))
```

Is this the kind of thing you were looking for?

Reply

1. Matthias Says:

July 5, 2010 at 7:44 pm

I love your last example -> Just great to see if you just know Java.

Go for Scala & functional programming!

4. links for 2010-08-18 « Dan Creswell's Linkblog Says:

August 18, 2010 at 12:06 pm

[...] Lots And Lots Of foldLeft Examples (tags: scala foldleft functionalprogramming) [...]

Reply

5. Kai Wähler Says:

August 31, 2011 at 12:25 pm

Hey Matt,

congratulations for your Scala blogs. I found this one using Google. It forwarded me to your foldLeft explanation. That one forwarded me to your Tail-Recursion basics. While reading them, Google

forwarded me to your “Null, null, Nil, ...” blog.

These blogs are very good, because they are natural and easy to understand! The main problem of Scala is the lack of good, easy to read blogs and books, still. Odersky and some others wrote books with good contents, but these books are only good as reference book, because they are too difficult to understand IMO.

Best regards,
Kai Wähler (Twitter: @KaiWaehner)

Reply

1. Matt Says:

August 31, 2011 at 2:38 pm

That's very kind of you, Kai. I'm glad you've enjoyed my posts. I've been meaning to write some new ones, and it's good motivation to know that people are reading them.

Reply

6. Jacek Says:

January 25, 2012 at 3:21 pm

Thanks for this great post! It provided a great way to get my hands dirty with the “foldLeft” way of thinking – which was a nice brain teaser and helped me understand the ideas behind (e.g. the recursive definition of lists in Scala).

Now I'm continuing to the rest of your Scala posts :)

Reply

7. bohtvaroh Says:

April 17, 2012 at 2:52 pm

Hello. You have a typo on your ‘contains’ function definition. Also it's worth saying that left fold will traverse all list even if the searching item is its head cause of Scala non-laziness. It's really impractical.

```
> def contains[A](xs: List[A], target: A): Boolean = xs.foldLeft(false)((contains, x) => {println(x);
contains || x == target})
> contains("asdfghjkk".toList, 's')
a
s
d
f
g
h
j
k
k
res18: Boolean = true
```

Reply

1. Matt Says:

April 17, 2012 at 9:27 pm

Thanks a lot, bohtvaroh! I fixed the typo.

Agreed, it's a completely impractical implementation. Most of these examples are not optimum solutions, but are just supposed to introduce the concept of list folding. I wish I could short-circuit the || logic in Scala. That's a good feature.

Reply

8. Ben Hardy (@benhardy) Says:

April 25, 2012 at 2:51 pm

Bravo, good stuff. Great examples.

Here's a simpler, more efficient way to calculate averages with fold by keeping tally of total and count and bringing them together via division afterward. I wouldn't be surprised if there's even more efficient ways than this.

```
def avg(numbers : Iterable[Double]) = {  
  val (sum, count) = numbers.foldLeft( (0.0, 0) ) {  
    (soFar, current) =>  
    (soFar._1 + current, soFar._2 + 1 )  
  }  
  sum / count  
}
```

Reply

9. From 2009 but lots of foldLeft examples [http...](http://oldfashionedsoftware.com/2009/07/30/lots-and-lots-of-foldleft-examples/) « Tyler Weir Says:

June 8, 2012 at 12:15 am

[...] 2009, but lots of foldLeft examples – <http://oldfashionedsoftware.com/2009/07/30/lots-and-lots-of-foldleft-examples/> Share this:TwitterFacebookLike this:LikeBe the first to like this post.
[...]

Reply

10. LeslieK Says:

January 28, 2013 at 8:12 pm

Thanks a lot. Really appreciated your effort in sharing your knowledge!

Reply

11. Stephen Boesch Says:

February 18, 2013 at 12:50 am

Here is a single pass version of the foldLeft

```
val lt = List(25,15,10,5,0)
var cnt=0; val avg = lt.foldLeft(0)((x,y) => cnt+=1;x+y) / cnt
```

Interesting note here: the compiler can not figure this out when we use underscores `_+_` : so need to do explicit `(x,y)`

Reply

12. Fold Left in Javascript (with all thinkable examples) | Marco Faustinelli's Muzietto Blog Says:

September 21, 2013 at 9:45 am

[...] [1] <http://oldfashionedsoftware.com/2009/07/30/lots-and-lots-of-foldleft-examples/> [...]

Reply

13. David Keen » Calculating distance with Scala's foldLeft Says:

October 17, 2013 at 8:43 pm

[...] was Matt Malone's page of lots and lots of foldLeft examples that put me on the right track, specifically his example for Average which showed how to use a [...]

Reply

14. Lovesh Harchandani Says:

April 20, 2014 at 6:34 pm

Thanks. I tried implementing the get function myself like this(my get function returns None if the index is out of bounds rather than raising an exception)

```
def get[A](lst: List[A], index: Int): Option[A] = // retrurns None for bad index
lst.foldLeft((-1, None))((r, c) => {
  if ((r._1+1) == index)
    (r._1+1, Some(lst(r._1+1)))
  else
    (r._1+1, None)
})._2
```

This gives me an error saying found Some[A] expected None.type for line `(r._1+1, Some(lst(r._1+1)))`. Now i understand that because my 1st parameter to foldLeft is a Pair `(-1, None)`, scala expects the second element of `(r._1+1, Some(lst(r._1+1)))` to be of None type but it finds Some. How can i correct this?

Reply

1. Matt Says:

April 21, 2014 at 2:36 pm

Hi Lovesh. This is a nice solution and the use of Option is an improvement over my original. I see two problems with your implementation. First, the compiler error is because scala can't figure out the type parameter "Option[A]" just based on the None in your first parameter.

Remember that foldLeft is declared as "def foldLeft[B](z: B)(f: (B, A) => B): B". The compiler has to figure out what B is. When you pass in `(-1, None)`, then scala decides that B is a `(Int, None)`. To make your code work, you have to tell the compiler that you want B to be more generic than `(Int, None)`. You want it to be `(Int, Option[A])`. Like this:


```
lst.foldLeft[(Int, Option[A])]((-1, None))((r, c) => {
```

The second problem is just a bug in your else expression. I think you want “(r._1+1, r._2)” instead of “(r._1+1, None)”.

Reply

1. Lovesh Harchandani Says:

April 21, 2014 at 3:24 pm

Thanks Matt. [(Int, Option[A])] was what i was looking for. Didn't know i could do that. And thanks for pointing the bug in my else expression

15. Link for implementation Says:

April 23, 2014 at 7:13 am

Hi

alternate implementation of some of your functions here

Reply

16. Hugh Says:

May 17, 2014 at 1:05 pm

Hi Matt great article I found it very helpful for revision for a Scala class I am taking!

Just one question: I don't understand why you use the :: operator and then a call to List.reverse(), when you could just use :+ which prepends.

For example, my code for your double function was:

```
def double(list: List[Int]): List[Int] = {
  list.foldLeft(List[Int]()) { (r, c) => r :+ c :+ c }
}
```

instead of your:

```
def double[A](list: List[A]): List[A] =
  list.foldLeft(List[A]())((r,c) => c :: c :: r).reverse
```

Reply

17. lovesh Says:

May 17, 2014 at 6:54 pm

Because Scala's List is a singly linked list so :+ operator has complexity O(n) so using :+ operator in each iteration for foldLeft will make the overall complexity of your example O(n^2). I wrote some of the examples here using :+ operator here wiredmonk.me/scala-foldleft-examples.html but at that time i didn't know this fact

Reply

18. rajeepprasanna Says:

Reblogged this on [solutions to the algorithm design manual](#).

[Reply](#)

19. Frank Says:

[September 3, 2015 at 8:50 pm](#)

Great stuff, thanks for sharing

[Reply](#)

[Create a free website or blog at WordPress.com.](#) — [The Connections Theme](#).