## LAB 3: MORE RECURSION WITH SCALA

In this lab you will practice further with recursion in Scala. You can use whichever of the tools introduced in Lab 1 you prefer to attempt the following exercises. You should use your lecture notes (part 4) and references listed under ‘Further reading’ to help you.

### Task 1. Lists and recursion

In Scala functional programming we think of lists as being composed of two parts:

* **head** – the first element in this list
* **tail** – the rest of the list (NOT the last element in the list!)

We also have the concept of an empty list, which we refer to as ***Nil***.

You can create and match lists using the *“cons”* syntax:

* Uses the **::** operator (actually it’s a method of the List class) that takes two arguments, a "head", which is a single element, and a "tail", which is a List
* A single element list is *“cons”*tructed from an element (head) and empty list (tail) – i.e. the tail is Nil

val single\_element\_list = 1::Nil

val longer\_list = 1 :: 2 :: 3 :: Nil

You can also create a list using the *range* method of List:

val x = List.range(1,6)

1. Include the following import statement so that you can use cons syntax:

import scala.collection.immutable.::

1. Use cons syntax to create a list *list1* with the values 1,2,3,4,5,6,7,8,9,10, and use the *range* method to create a list *list2* with the same values. Declare *list2* as a *var* rather than a *val*.
2. Write an expression that joins the two lists with the **::** operator. Look carefully at the result. Write another expression that uses the **:::** operator instead. Look carefully at the result of this, and deduce the difference between the purpose of these operators. Write another expression that uses the ++ operator instead. Which of the other operators does this behave like? (Scala often seems to have more than one way of doing the same thing!)
3. The List type has methods to get the head and tail of a list, so you can easily split a list into its head and tail. The head is a single element, the tail is another list (which may be empty). However, you need to do a little bit more work to find the last element of a list.   
     
   You can get the last element using recursion – you split the list initially into head and tail. If the tail is empty, then the head of the list is the last element. If not, you repeat exactly the same process with the tail. You keep repeating this (recursion) until you have a list with an empty tail (stopping condition) and then return the head of that list.
4. Test the following function that does this by writing the function and calling it on *list1*.

def **lastif**(ls:List[Int]):Int = {  
 if(ls.tail == Nil)  
 ls.head  
 else  
 **lastif**(ls.tail)  
}

1. Test the following version of the same function that uses pattern matching instead of an if expression (you may get a warning about exhaustiveness with this one, ignore that for now). Which style do you prefer?

def **last**(ls: List[Int]): Int = ls match {  
 case h :: Nil => h  
 case \_ :: tail => **last**(tail)  
}

1. Write a function *sum* that uses recursion to calculate the *sum* of the elements of a list of Ints. You can use either an *if* expression or a *match* expression. What will the stopping condition be in this case - It’s actually slightly simpler than the length example?

Note that List has a method *sum* that does the same job as your sum function – test this and check that it gives the same result. It is generally better to use the built-in method where available, but it is useful to understand how to use recursion on a list if you need to do something that is not already available.

### Task 2 Tail recursion

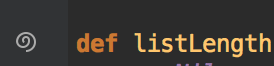
In this task you will continue to use the lists you created in Task 1.

1. Test the following function that calculates the length of a list. You can call the function to get the length of *list2*.

def listLength(ls:List[\_]):Int = ls match{  
 case Nil => 0  
 case h :: tail => 1 + listLength(ls.tail)  
}

(this can also be written with an *if* expression – see listing at end)

If you are using a Scala Worksheet in IntelliJ, note the symbol next to the function name. Hover the cursor over this for an explanation.



1. Add the following statement before your call to *listLength*. This builds *list2* into a much longer list.

1 to 15 foreach(x=> list2 = list2++list2)

What happens? Why?

1. Replace *listLength* with the following tail-recursive version and repeat the test.

def listLength(ls:List[\_]):Int = {  
 def listLength\_nested(ls:List[\_], len: Int):Int = ls match{  
 case Nil => len  
 case h :: tail => listLength\_nested(ls.tail, len+1)  
 }  
 listLength\_nested(ls, 0)  
}

(this can also be written with an *if* expression – see listing at end)

What happens now? What symbol do you see next to the function name in IntelliJ, and what explanation is given when you hover over it?

1. Write and test a tail-recursive version of your *sum* function
2. Write and test a tail-recursive version of the recursive *power* function you created in Lab 2.
3. (Challenge) The following function (if and match versions shown) calculates the nth element of the Fibonacci sequence 1,1,2,3,5,8,13,… (each element is the sum of the two previous ones)

def fib(n: Int): Int = n match {  
 case 0 => 0  
 case 1 => 1  
 case \_ => fib(n - 1) + fib(n - 2)  
}

def fib(n: Int): Int = {  
 if (n <= 1)  
 n  
 else  
 fib(n - 1) + fib(n - 2)  
}

So, *fib(6)* gives the result 8.

This is not tail-recursive. Write a tail-recursive version of this function.

Versions of *listLength* function using *if* expressions:

def listLength(ls:List[\_]):Int = {  
 if(ls == Nil)

0  
 else

1 + listLength(ls.tail)  
}

def listLength(ls:List[\_]):Int = {  
 def listLength\_nested(ls:List[\_], len: Int):Int = {  
 if(ls == Nil) len  
 else listLength\_nested(ls.tail, len+1)  
 }  
 listLength\_nested(ls, 0)  
}