COMS W3137 - Recitation Notes
Week 3: Functional Programming Basics

Daniel Bauer

February 4, 2022

1 `for` loops and expressions

1.1 Iteration with Side Effects

We have previously discussed while loops. While loops require updates to a `var`, so that the looping condition can eventually become false. Scala also support `for` loops. At first glance, `for` loops behave like enhanced `for` loops in Java: We can iterate over the items in an iterable data structure. This does not require `var`s, but for loops always have side effects, such as printing output or adding to a mutable data structure.

```
scala> val li = List(1,2,3,4,5)
val li: List[Int] = List(1, 2, 3, 4, 5)

scala> for (x : Int <- li)
     |     println(x * 2)
2
4
6
8
10

scala> var newList : List[Int] = List[Int]()
var newList: List[Int] = List()

scala> for (x : Int <- li)
     |     newList = newList :+ x * 2

scala> newList
val res0: List[Int] = List(2, 4, 6, 8, 10)
```

1.2 `for` comprehensions

`for` loops can also be used as expressions, using the `yield` keyword. This is called a *comprehension*. When we use a `for` comprehension on a list, the expression evaluates to a list (of the same generic type), without requiring any explicit change of state.

```
scala> for (x : Int <- li) yield (x * 2)
val res0: List[Int] = List(2, 4, 6, 8, 10)
```

The `yield` keyword tells Scala to aggregate the result if the expression `x  2` in a new list. We can also include a filter. Only values that pass the filter condition are included in the resulting list.

```
scala> for (x : Int <- li; if x \% 2 == 0) yield (x * 2)
val res0: List[Int] = List(4, 8)
```

Also works: for (x : Int <- li) yield (if (x%2 == 0) x * 2 else x)

Exercise: Without `for`-comprehensions or some of the other abstractions discussed below, iterating through a list in a side-effect free and expression oriented way would be tedious: It would require recursion. Write a function `def double(li : List[Int]) : List[Int]` that uses only recursion (no `while` or `for` loops allowed) to construct a new list in which each integer has been doubled.

NB: \`::\` creates a new list, whereas \`:+\` appends to an existing list

2 map, filter

Rather than having to traverse data structures using recursion, or using a `for` loop, data structures such as lists provide a `map` method. This method takes a *function object* as its parameter, applies it to each element in the data structure and returns a new data structure.

```
scala> def timestwo(x : Int) : Int = x * 2

scala> li.map(timestwo)
res0: List[Int] = List(2, 4, 6, 8, 10)
```

Also works: li.map((x : Int) => x * 2)

A related method, `filter`, returns a new collection that includes only the elements for which a certain condition is true. Like map, `filter` takes a function object as a paremeter. This function object must return a boolean. This function is applied to each element in the collection. If the function return true, the element is included in the result, otherwise it is skipped.

```
def isEven(x : Int) = x % 2 == 0
isEven: (x: Int)Boolean

scala> li.filter(isEven)
res1: List[Int] = List(2, 4)
```

Also works: li.filter((x : Int) => x % 2 == 0)

Can chain these, ie: li.filter((x : Int) => x % 2 == 0).map((x : Int) => x + 2)

Can simplify more by not including type annotation: li.filter(x => x % 2 == 0).map(x => x + 2)

The `map` and `filter` method are defined in the trait `scala.collection.TraversableLike`, which is extended by almost all scala data structures.
`map` and `filter` are examples of higher-order functions. A `higher-order function` is any function that either expects a function object as a parameter, or returns a function object.

3 Function literals / `val` function

So far we have defined functions with `def`. Unless `def` is used in the REPL, such definitions always appear in the context of a class or object. As a result, `def` is used to define *methods*. A method always has an implicit reference to the class instance or object it is called on.
An alternative way to define a function is as a *function literal*, using `=>`.

```
scala> val incr = (x : Int) => x + 1 : Int
incr: Int => Int = $$Lambda$$1218/0x0000000800781840@1774c4e2
```

Function literals are also called *lambdas* in other languages, including more recent versions of Java (≥ 1.8). They are often used to define a small function in-place.
The `(x : Int)` on the left hand side of the `=>` specifies the parameter list of the function. The `: Int` at the end of the line specifies the return type. In many cases, we do not have to make the return type explicit and Scala's type inference can infer it automatically. `x+1` is the function body, which can be any expression involving the parameters. The *type* of the function `incr` is `Int=>Int`. The function takes an integer as a parameter and returns an integer.
Once the funciton is defined, we can call it as usual.

```
scala> incr(5)
res4: Int = 6
```

3.1 Partially applied functions

A *partially applied* function is a function in which some of the arguments have already been provided. Consider for example the following function `add`.

```
scala> val add = (x : Int, y : Int) => x + y
add: (Int, Int) => Int = $$Lambda$$1176/0x000000080074f840@5fffb692
```

We can now provide only one of the parameters, leaving the other one to be completed later.

```
scala> val add2 = (x:Int) => add(x,2)
add2: Int => Int = $$Lambda$$1233/0x0000000800780040@63636de0

scala> add2(4)
res0: Int = 6
```

If the parameters of the function appear in the same order in the function body, the following notation can be used as a short-cut in function literals. This simplifies the definition of partially applied functions:

```
scala> val add2 = add(_,2)
add2: Int => Int = $$Lambda$$1240/0x0000000800784840@3ed87b6e
```

We can also use this technique to convert `def` methods in to `val` functions. This process is also called *eta expansion*.

```
scala> def add(x : Int, y : Int) = x+y
add: (x: Int, y: Int)Int

scala> val addToo = add(_,_)
addToo: (Int, Int) => Int =
    $$Lambda$1242/0x0000000800786040@55e3b64d
```

A single trailing _ can be used to replace the entire parameter list of a function.

```
scala> val addToo = add _
addToo: (Int, Int) => Int =
    $$Lambda$1242/0x0000000800786040@55e3b64d
```

3.2 Currying

Typically, when a function is called, all arguments to the function need to be provided. It is also possible to define a function that accepts only some of its parameters at a time. Such a function is called a `curried` function. [1] Technically, a curried function is a higher-order function because it returns another function.

[1] Currying functions are named after the American mathematician Haskell Curry, but also called Schönfinkeling after the Russian mathematician Moses Schönfinkel

```
scala> val curriedAdd = (x : Int) => { (y : Int) => x + y }
curriedAdd: Int => (Int => Int) =
    $$Lambda$$1395/0x0000000800838040@1f7c18e5

scala> curriedAdd(1)
res39: Int => Int = $$Lambda$$1396/0x0000000800839040@61ac4f69

scala> curriedAdd(1)(2)
res40: Int = 3
```

Note that the `x` in the nested function `(y : Int) => x+y` is the parameter of the surrounding function. Holding on to the local vairables of a function in a nested function is called a *closure*. Closures appear commonly in functional programming as a way to store program state. The state is preserved in the function object that is returned from the outer function.
For `def` methods there is special notation to define a curried method by specifying multiple parameter groups. This does not work with function literals.

```
scala> def curriedAdd(x :Int)(y:Int) = x+y
curriedAdd: (x: Int)(y: Int)Int

scala> curriedAdd(1)(2)
res0: Int = 3
```

We can verify that passing only one parameter actually returns a method (which we need to eta convert to a function):

```
scala> val f : (Int => Int) = curriedAdd(1) _

scala> scala> f(2)
res4: Int = 3
```

Or we can eta expand the entire method:

```
scala> curriedAdd _
res6: Int => (Int => Int) =
    $$Lambda$1141/0x0000000800737840@2c08c787
```

Exercise: Write a function `curry3(f : (Int,Int,Int) => Int) : (Int=>(Int=>(Int=>Int)))` that returns a curried version of the function f. For an additional challenge, can you make the function generic so that it works for any function with three parameters (regardless of their type)?

3.3 Example: Function Composition and Repeated Application

Given two mathematical functions f and g, the function composition $f \circ g$ (*"g following/afer f"*) produces a function $h(x) = f(g(x))$. In other words, the function h first computers $g(x)$ and then applies f to the result. Assume we are given functions `f : Int=>Int` and `g : Int=>Int`. Write a function `combine: (Int=>Int, Int=>Int) => (Int=>Int)`. Calling `combine(f,g)` should return a function `h: Int=>Int`, so that $h = f \circ g$.

solution:

```
val combine = (f : Int=>Int, g:Int=>Int) => (x:Int) => f(g(x))
```

or using a polymorphic (generic) function (note that there are no polymorphic function literals in Scala):

```
def combine[A,B,C](f : A=>B, g: B=>C) = (x:A) => g(f(x))
```

Now assume you wish to apply the same function k times. Given a function f and a positive integer k, the k-th repeated application of k is the function $h(x) = f(f(\cdots(f(x))\cdots))$. For example, the 2nd repeated application of the function $f(x) = x + 1$ is $h(x) = f(f(x))$ and $h(2) = 3$. We wish to define a Scala function `repeat(k: Int, f: Int=>Int, k: Int): Int=>Int` that returns the`k-th` repeated application of `f`.

solution: We can use recursion and the `combine` function we just defined.

```
val repeat : ((Int=>Int, Int)=>(Int=>Int)) = (k : Int, f:
    Int=>Int) => k match {
    case 1 => f
    case _ => combine(f, repeat(k-1, f))
}
```

Interestingly, we can use also define the repeat function as a curried function, like this:

```
val repeat : ((Int=>Int, Int)=>(Int=>Int)) = (k : Int, f:
    Int=>Int) => k match {
    case 1 => f
    case _ => combine(f, repeat(k-1, f))
}
```

This example illustrates how curried functions can be used as control structures.

```
scala> repeat(3){ (x : Int) => x+1 } (1)
res20: Int = 4
```

4 foldLeft, foldRight, fold

Recall the `map` and `filter` methods described above. One thing that these methods have in common is that they separate the traversal through the list from the operation performed on the list element. We can generalize this idea even further.
Consider the task of computing the sum of a list. We could do this recursively by first computing the sum of the `tail` list, adding the head, and returning the value. In the base case, we are calling the recursive function on an empty list `Nil` and return 0.

```
scala> def sumRec(li : List[Int]) : Int = li match {
     |      case Nil => 0
     |      case ::(head, tail) => sumRec(tail) + head
     | }
sumRec: (li: List[Int])Int

scala> sumRec(List(1,2,3,4,5))
res0: Int = 15
```

We can use the same technique to compute all kinds of things on the list. We only need to specify 1) the return value for the base case and 2) a way to combine the return value of the recursion with the next element.
The `foldRight` method defined on the list allows you to do exactly that.
`foldRight` is a curried method with two parameter groups. The first one specifies the base case value. The second one specifies a function object that combines the return value and the next head to produce a new return value, as in the example before.
Here is the signature of that method:

```
foldRight[A,B](base_case : A)(combiner : (B,A)=>A)
```

Using fold, we can rewrite the sum function as follows.

```
scala> def sum(li : List[Int]) = li.foldRight(0)((x:Int, y:Int) =>
    x+y)
sum: (li: List[Int])Int

scala> sum(List(1,2,3,4,5))
res1: Int = 15
```

Also works: li.foldRight(0)((x, y) => x + y)

Interestingly, both map and filter can be defined using `fold`.

li.foldRight(0)((head, tailresult) => head + tailresult)

foldLeft starts on the left side—the first item—and iterates to the right; foldRight starts on the right side—the last item—and iterates to the left

```
scala> def myMap[A,B](li : List[A], f: A=>B) : List[B] =
     | li.foldRight(Nil : List[B])((next_head : A, new_tail :
    List[B]) =>
     |      f(next_head) :: new_tail)
myMap: [A, B](li: List[A], f: A => B)List[B]

scala> myMap(List(1,2,3,4,5), (x :Int )=>x+1)
res2: List[Int] = List(2, 3, 4, 5, 6)

scala> def myFilter[A](li : List[A], f: A=>Boolean) : List[A] =
     | li.foldRight(Nil : List[A])((next_head : A, new_tail :
    List[A]) =>
     |        if (f(next_head)) next_head :: new_tail else new_tail)
myFilter: [A](li: List[A], f: A => Boolean)List[A]

scala> myFilter(List(1,2,3,4,5), (x :Int )=>x%2==0)
res3: List[Int] = List(2, 4)
```