Functional Programming: More on Functions

This lesson introduces a testing tool largely used with Scala. Afterwards some advanced topics on functions will be covered.

1. ScalaTest

ScalaTest is a very popular tool used to test Scala code. It supports different styles of testing through the use of distinct style traits. FunSuite is just one of them and it will be used in this document. The style affects only how the declarations of the tests will look, and assertions and everything else are independent from the style.

To start, create a new “Scala “> “sbt” project in Intellij IDEA and name it “firstTestProject”.

Add ScalaTest to your project by adding the following to your build.sbt file in the root of the project.

```
libraryDependencies += "org.scalactic" %% "scalactic" % "3.0.5"
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.5" % "test"
```

Afterwards Intellij IDEA will notice that “build.sbt” was changed and ask what to do. You should “Refresh project”.

To make sure everything works, we will run a simple test.

In *src/main/scala*, create a *tap* package. Inside add a Scala Object, which is done by adding a Scala Class and then choosing it to be an object. Create the *removeDigits* function as follows:

```
package tap
object StringFunctions {
 def removeDigits(s: String): String = s.filter(!"0123456789".contains(_))
 def incDigits(s: String,i: Int) = s.map(x => if (x.isDigit) (x+i).toChar else x)
}
```

To create a test for *StringFunctions* object you must place the cursor inside the object and press CTRL+SHIFT+T. Choose “Create New Test...” and the ScalaTest testing library, additionally select the *removeDigits* function.

Now there should be a *StringFunctionsTest* class in *src/test/scala/tap*. Change it to the following:

```
package tap
import org.scalatest.FunSuite

class StringFunctionsTest extends FunSuite {

 test("removeDigits on a blank string should result in a blank string") {
  assert(StringFunctions.removeDigits("")==="")
 }
```

```
 test("removeDigits on abc95def0ghi should result in abcdefghi ") {
  assert(StringFunctions.removeDigits("abc95def0ghi")==="abcdefghi")
 }
 test("removeDigits on a null should throw an exception") {
  val e = intercept[NullPointerException] {
   StringFunctions.removeDigits(null)
  }
  assert(e != null)
 }
}
```

Besides `assert` for general assertions, two other assertions can be used with ScalaTest: `assertResult` to distinguish expected from concrete values and `intercept` to ensure a piece of code throws an expected exception.

The triple equals operator used in the `StringFunctionsTest` class provides more informative error message.

The tests can be run by invoking the “sbt shell” in View > Tool Window > sbt shell. In the shell, you can issue the command **compile**, which compiles the project. You can also issue the **test** command which tests all the project. The tests can be run from the IDE or the sbt shell. If run directly

There is also the **console** command. This command loads the project and waits for commands in a scala command prompt. Any function of the project can be executed this way. Let’s try to execute `tap.StringFunctions.removeDigits("a1b2c3")` the result should appear as `"abc"`. The main feature of the console command is that you can interactively execute any command inside your application. Try it fir the `tap.StringFunctions.incDigits` function.

The console mode can be abandoned with CTRL-d command. To terminate the sbt shell there is the **exit** command. After the first execution of the “sbt shell” the tests can be run by simply executing the test file in the user interface.

2. Higher Order Functions

Functions are values and can be assigned to variables and passed as arguments to functions. Functions that take other functions as parameters or that return functions as results are called higher-order functions.

The function `resultToString` is a higher-order function as it accepts a function as its third argument (`f: Int=>Int`).

```
def resultToString(name: String, n: Int, f: Int=>Int): String = {
 val msg = "The %s of %d is %d."
 msg.format(name, n, f(n))
}

def double(x: Int) =x*2

def triple(x: Int) =x*3
```

```
def quadruple(x: Int) =x*4

assert(resultToString("double", 5, double)=="The double of 5 is 10.")
assert(resultToString("triple", 5, triple)=="The triple of 5 is 15.")
assert(resultToString("quadruple", 5, quadruple)=="The quadruple of 5 is
20.")
```

The type A => B is the type of a function that takes an argument of type A and returns a result of type B. So, `Int=>Int` is the type of functions that map integers to integers.

Exercise

Using the project *firstTestProject*, in package *tap*, create the following Scala object:

```
package tap

object HigherOrderFunction {

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

 def isOdd(x: Int) = x % 2 != 0

 def square(x: Int) = x * x

 def cube(x: Int) = x * x * x

 def isPrime(x: Int) = {
  ???
 }

 def  hof( f :Int => Boolean, m : Int => Int, xs :List[Int] ): List[Int] =
 {
  ???
 }
}
```

Function `hof` is supposed to receive a filter function `f` and mapping function `m`. The function `isPrime` is a predicate that checks weather a number is prime or not. Complete the functions and create a test (using CTRL-SHIFT-T), making sure the code of the tests is like the following:

```
package tap
import org.scalatest.FunSuite
import HigherOrderFunction._

class HigherOrderFunctionTest extends FunSuite {

 test("testIsPrime") {
  val l = (1 to 30).filter(isPrime(_))
  assert( l === List(1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29))
 }

 test("testHof") {
  val l = (1 to 5).toList
  assert( hof( isEven, square, l ) === List(4, 16) )
  assert( hof(  isOdd,   cube, l ) === List(1, 27, 125) )
  assert( hof(isPrime, square, l ) === List(1,  4,  9,  25) )
 }
```

```
 test("isEven") {
  assert(( 1 to 15 ).filter( isEven ) === List(2, 4, 6, 8, 10, 12, 14))
 }

 test("isOdd") {
  assert(( 1 to 15 ).filter( isOdd ) === List(1, 3, 5, 7, 9, 11, 13, 15))
 }
 test("square") {
  assert(( 1 to 6 ).map( square ) === List( 1, 4, 9, 16, 25, 36))
 }

}
```

The functions will be correct when all the tests pass.

3. Anonymous Functions

Passing functions as parameters can lead to the creation of many small functions. In fact these functions do not need to be defined as seen before using `def`.

Considering the function `abs` it is not necessary to define a variable to be used as argument to `abs` (`val i=5`; `abs(i)`), but a value can be passed directly to the function (`abs(5)`).

Analogously it is possible to write a function without giving it a name, i.e. an anonymous function. Thus, a function literal or an anonymous function is a function with no name in Scala source code, specified with function literal syntax. For example, `(x: Int, y: Int) => x + y` is an anonymous function that sums two integers and returns the result (an `Int`).

Exercise

We shall now change the tests in HigherOrderFunctionTest to use anonymous functions:

```
test("isOdd") {
 assert(( 1 to 15 ).filter( ??? ) === List(1, 3, 5, 7, 9, 11, 13, 15))
}

test("square") {
 assert(( 1 to 6 ).map( ??? ) === List( 1, 4, 9, 16, 25, 36))
}
```

Replace each `???` in order to have anonymous functions in the calls.

4. Currying

Currying is a way to write functions with multiple parameter lists and it is a powerful functional programming technique. For instance, `def f(x: Int)(y: Int)(z:String)` is a curried function with three parameter lists. A curried function is applied by passing several arguments lists, as in: `f(3)(4)("TAP")`.

A curried function is applied to multiple argument lists. Example of a non-curried function that adds two Int parameters, x and y:

```
def nonCurriedSum(x: Int, y: Int) = x + y
```

Here is a similar function, but curried, with two lists of one Int parameter each:

```
def curriedSum(x: Int)(y: Int) = x + y
```

The call to these functions (`nonCurriedSum(3,6)`) and (`curriedSum(3)(6)`) provide the same results (`9`). You can use the placeholder notation to use curriedSum in a partially applied function expression without explicitly indicating the second parameter:

```
def sum2=curriedSum(2)_                    //> sum2: => Int => Int
def sum3=curriedSum(3)_                    //> sum3: => Int => Int
sum2(3)                                    //> res4: Int = 5
sum3(3)                                    //> res5: Int = 6
```

Exercise

Consider the following function:

```
def multiply(a:Int)(b:Int):Int = a * b
```

Provide the definition of two curried functions `multiply_by_2` and `multiply_by_3` and use the placeholder notation to do not have the second parameter explicitly indicated. Use ScalaTest to test the functions.

5. Consolidating everything

In the section a set of exercises is proposed to provide a functional representation of sets of integers based on the mathematical notion of characteristic functions.

Mathematically, the function which takes an integer as argument and which returns a `boolean` indicating whether the given integer belongs to a set is called the characteristic function of the set. For example, we can characterize the set of negative integers by the characteristic function `(x: Int) => x < 0`.

Therefore, we choose to represent a set by its characteristic function and define a type alias for this representation:

```
type Set = Int => Boolean
```

Using this representation, we define a function that tests for the presence of a value in a set:

```
def contains(s: Set, elem: Int): Boolean = s(elem)
```

Download **lab02.zip** from moodle and examine the two files. One of them is a ScalaTest file to be used to test your functions and the other has some functions that needed to be completed by substituting the expressions "???". Add the files to your project in the tap package.

Exercises: Basic Functions on Sets

1. Define a function which creates a singleton set from one integer value: the set represents the set of the one given element. Its signature is as follows:

```
def singletonSet(elem: Int): Set
```

2. Define the functions `union`, `intersect`, and `diff`, which takes two sets, and return, respectively, their union, intersection and differences. `diff(s, t)` returns a set which contains all the elements of the set s that are not in the set t. These functions have the following signatures:

```
def union(s: Set, t: Set): Set
def intersect(s: Set, t: Set): Set
def diff(s: Set, t: Set): Set
```

3. Define the function `filter` which selects only the elements of a set that are accepted by a given predicate `p`. The filtered elements are returned as a new set. The signature of `filter` is as follows:

```
def filter(s: Set, p: Int => Boolean): Set
```

Exercises: Queries and Transformations on Sets

1. The function `forall` tests whether a given predicate is true for all elements of the set and has the following signature:

```
def forall(s: Set, p: Int => Boolean): Boolean
```

   Note that there is no direct way to find which elements are in a set. `contains` only allows to know whether a given element is included. Thus, if we wish to do something to all elements of a set, then we have to iterate over all integers, testing each time whether it is included in the set, and if so, to do something with it. Here, we consider that an integer x has the property -1000 <= x <= 1000 in order to limit the search space.

   Implement `forall` using linear recursion. For this, use a helper function nested in `forall`. Its structure is as follows (replace the ???):

```
def forall(s: Set, p: Int => Boolean): Boolean = {
 def iter(a: Int): Boolean = {
  if (???) ???
  else if (???) ???
```

```
  else iter(???)
 }
 iter(???)
}
```

2. Using `forall`, implement a function exists which tests whether a set contains at least one element for which the given predicate is true. Note that the functions `forall` and `exists` behave like the universal and existential quantifiers of first-order logic.

```
def exists(s: Set, p: Int => Boolean): Boolean
```

3. Write a function map which transforms a given set into another one by applying to each of its elements the given function. `map` has the following signature:

```
def map(s: Set, f: Int => Int): Set
```