

# Lab : Apache Spark Currying and Partially Applied Functions



## Pre-reqs:

- Google Chrome (Recommended)

## Lab Environment

## Function currying in Scala

Function currying in Scala is used to create partially applied functions. The curried functions are declared with multiple parameter groups with each group enclosed in paranthesis.

For example,

```
def sum(a: Int) (b:Int): Int = {  
  > a + b  
  > }
```

There can also be multiple parameters in each parameter group. We can then use this curried function and create partially applied functions.

## Partially applied functions in Scala

The partially applied functions as the name suggests are applied partially by only passing some parameters while holding back other parameters. The partially applied functions are used when you do not want to pass all the parameters at once but instead, you can pass some parameters first and then the other parameters at later time.

For example, we can partially apply the function which we created above using currying.

```
val part = sum(54) _
```

This will return us a function object called part.

We can now pass the parameter which we held back as shown below.

```
part(6)
```

## Task 1: Defining Currying Functions

Let's have a look and currying functions. We shall be working in the Scala shell for this and the next task.

**Step 1:** Open the terminal and fire up the Scala shell by simply typing in Scala in the terminal. You should see the Scala prompt.

```
scala
```

**Step 2:** Let us now define a simple currying function with two parameter groups as shown below to understand the concept of curried functions.

```
def sum(x: Int) (y: Int): Int = {  
  | x + y  
  | }
```

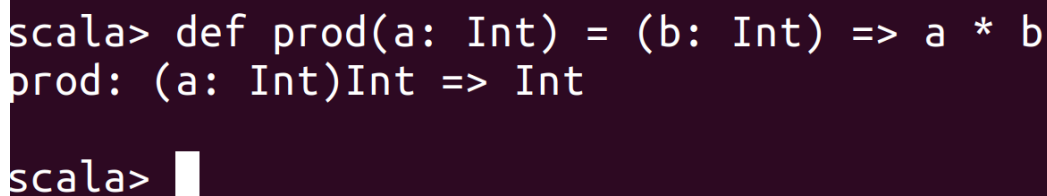
We have defined a function called sum which adds two numbers. Instead of passing the parameters as one group, we have curried the parameters in two parameter groups. This will help us with partially applied functions.

**Step 3:** We can also define currying functions with multiple parameters inside each parameter group as shown below.

```
def sumProd(a: Int, x: Int) (b: Int, y: Int): Int = {  
  | a * b + x * y  
  | }
```

**Step 4:** We can also define a currying function in such a way that we can transform a function which takes two or more parameters into a function that takes only one parameter.

```
def prod(a: Int) = (b: Int) => a * b
```

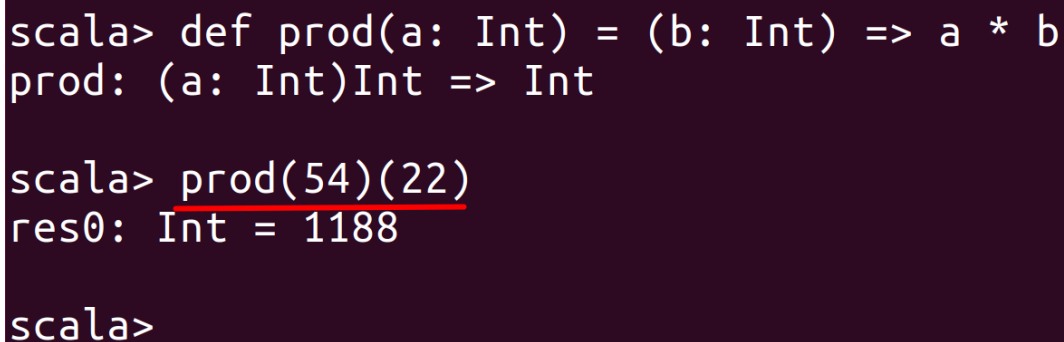


```
scala> def prod(a: Int) = (b: Int) => a * b  
prod: (a: Int)Int => Int  
scala> 
```

As you can see from the screenshot above, we have declared a prod function which only takes one parameter a and returns another function which in turn takes another parameter b and returns the result.

**Step 5:** We can simply pass the arguments with each argument inside a paranthesis as shown below.

```
prod(54)(22)
```



```
scala> def prod(a: Int) = (b: Int) => a * b  
prod: (a: Int)Int => Int  
  
scala> prod(54)(22)  
res0: Int = 1188  
  
scala> 
```

Task is complete!

## Task 2: Using partially applied functions

Let us now see how we can use the curried function and apply them partially.

**Step 1:** We have created a sum function in step 1 of previous exercise. Let us use that function and partially apply the parameters for that function.

```
val sumObj = sum(6)_
```

This will return us a function object as shown in the screenshot below.

```
scala> val sumObj = sum(6)_  
sumObj: Int => Int = <function1>  
  
scala> sumObj(5)  
res1: Int = 11  
  
scala> 
```

The `_` is used as a placeholder for the parameter we are holding back. It indicates the compiler that we are partially applying a function.

**Step 2:** We can then use the function object later to pass the parameter which we held back as shown below.

```
sumObj(5)
```

**Step 3:** Similarly, let us partially apply the `sumProd` function which we created in the step 2 of previous task.

```
val sumProdObj = sumProd(5, 6)_
```

We can then pass the held back parameters at a later time as shown below.

```
sumProdObj(7, 8)
```

```
scala> val sumProdObj = sumProd(5, 6)_  
sumProdObj: (Int, Int) => Int = <function2>  
  
scala> sumProdObj(7, 8)  
res2: Int = 83  
  
scala> 
```

The result is as shown in the screenshot above.

Task is complete!