Exercise 2 - Syntax Basics

Objective

The objective behind this exercise is to get some experience with basic language features in scala and experiment with what we have seen in the slides so far. We also will be looking at writing for comprehensions and using recursion to solve problems.

References:

Slides: 25-48

Overview

Open a new worksheet for these exercises. This will help you see the program develop and allow you to refer back to previous parts of the exercise. The worksheets are a ‘work in progress’ both for eclipse and IntelliJ. They will sometimes not recognise valid scala. If you are sure something should work then copy the relevant line to the REPL and see if it is a bug you’ve found. At this stage you shouldn’t find many bugs, but as the course progresses, especially with objects, the work sheets can sometimes cause some issue.

1. Create a new val and var, observe the types returned from the expressions.

val n = 5

var m = 5.0

1. Reassign the values for n and m to be 10.0 and 20 respectively. Does it work?

Vals are ‘immutable’ objects. Their values can’t change. We can change the value of the m to anything we like as long as the type doesn’t change, but once a val is declared it will always hold the same value.

1. Create some more vals in the worksheet. Look at the type returned for each of these. The compiler will infer the type where it is able.

You should already have val n = 5; var m = 5.0; and a reassignment for m.

val name = "Alice"  
val c = 'a'  
val f = 3.0f  
val b = true

1. Types can be converted between. Try running the following statements and see what the results are.

name + c  
c + n  
name + n  
name + b  
b & false  
b | false  
f + m  
f + m + n  
n + f

When you add strings and characters together you get a different result from when you add a character and a number, or a string and a number. Characters can be interpreted as numeric references (it uses ASCII to determine what number the character is). Adding strings and numbers together will output another string. Always make sure of your type when calling methods on objects.

1. It is considered good practice to let the compiler infer type for scala expressions. Sometimes you will need to override this. We can do that when the val is declared. Declare these two vals and see what happens when you try change the value between 10, 10.0 and 10f.

val thisIsADouble : Double = 10;  
val thisIsAFloat : Float = 10;

1. Statement blocks in code use parenthesis to group statements together { … }

An example of this is:

val result = {val a = 20; val b = 5; a/b}

Statement blocks can be on more than one line, the semi-colons are only needed when more than one statement is on the same line. The last line in the block is the return value.

Create some statement blocks to evaluate the following (you should create the vals required inside the statement blocks):

* + y = m \* x + c where m is 1.0, x is 10 and c is 15
  + total = input + (input \* 20%) where the input is 10.00
  + fullname = first name space surname
  + a = 5, b = 10, print out a + b

What is the return type of the final block?

Control Flow

An if statement in scala has the form:

if (c == 'a') *println*("character is a")  
if (c != 'a') *println*("character is not a")

These are similar to java if statements.

1. Write an if-statement that says if a given year is a leap year or not. The rules for if something is a leap year are:
   * Year is divisible exactly by four and
   * Year is not divisible exactly by 100 or year is divisible exactly by 400

Create a function which uses your if statement with a given parameter

1. Write a for-comprehension that tests each of the years from 1960 to the present to see if they are leap years.

Hint: Look up ranges to see how to go from one date to another. You can also call your code from question 7

1. Write another for comprehension that prints out every letter in hello with stars in between

For example: \*h\*e\*l\*l\*o\* \*w\*o\*r\*l\*d\*

1. Write a for comprehension with a guard that yields a list of all the numbers between 1 and 100 that are even

Recursion

A recursive statement is one that calls itself. These are very important in functional programming where we do not have the traditional while and for loops (notice that we use for comprehensions not for loops!)

When writing a recursive statement always make sure you know what the end point for the recursion is and write the test for this first. Then write the recursive step in a program. To store an accumulated value in each step of the recursion we can use an *accumulator.* This is a value passed through each step of the recursion and added to (or multiplied by) each time.

In scala recursion is the only time it is mandatory to have a return type for a function as it cannot be inferred due to Object Oriented languages ability to create sub classes.

1. Write a recursive statement to calculate the factorial of a number. The factorial is calculated using the following

factorial(n) = n \* (n-1) \* (n-2) \* … \* 1;

For example, the factorial of 5 is calculated as:

factorial(5) = 5 \* 4 \* 3 \* 2 \* 1 = 120

You will need an accumulator to hold the current value. Remember to work out the end point of your recursion before you start!

1. Write a function to calculate x to the power of n (x^n), using this recursive definition:
   * If n is even and greater than 0, we have two steps:
     + y = x ^ (n/2)
     + return y \* y
   * If n is odd and greater than 0, x^n = x \* (x^(n-1))
   * If n is 0, return 1
   * If n is less than 0, x^n = 1 / (x^(-n))

Don't use a return statement, and don’t worry about the maths behind this too much. We are interested in using recursion with an if-statement, not whether this is the most efficient way of doing it!

The symbol ^ is a method name for the Int object. You may want to name your method something else (such as power(x,n) )

SBT Continuous compile / integration loop

For this exercise we are going to look at sbt again and how we can have a continuous compile and run loop with out code.

Open a terminal window and create a new directory for SBT, change to that directory. Open a text editor and create a new file. In the file add the skeleton we used in exercise one to print “Hello World” Save this file in the same directory as you created for sbt earlier.

object hello extends App {

println("Hello World")

}

Now in run sbt from the command line with ‘sbt’ In the sbt window we can start a continuous loop by using the tilde key ~

In sbt, type ~ compile:run

This will start the compile and run loop. The commands are executed one after another each time a change in the source code is detected. Change your file in the text editor and press save. You should see sbt compile and run the new file every time you save it.

To end the continuous loop, press enter in the sbt window.