IP Lecture 1: Introduction

Joe Pitt-Francis
—with thanks to previous lecturers especially
Mike Spivey (most course materials) & Gavin Lowe (Scala adaptations)—

Overview

This is a course about programs written in an imperative style.

In functional programming, programs have no state: a program is formed by applying mathematical functions to arguments.

By contrast, programs written in an imperative style store values in variables. The values of these variables can change as the program proceeds.

The big picture

Functional programming (last term with Geraint Jones)

Programs as mathematical functions. Applying mathematical reasoning to programs to prove programs correct.

Imperative programming 1 (this term)

Programming with state. Reasoning about loop-based programs with some examples from "Design and analysis of algorithms".

Imperative programming 2 (this term)

Data structures and encapsulation. Specifying, programming and correctness with abstract datatypes.

Imperative programming 3 (next term with Pete Jeavons)

Programming in the large. Object-oriented techniques and design patterns.

Course aims

Part one. To learn:

- how to program in an imperative style;
- how to reason mathematically about imperative programs, particularly programs that use loops;
- how to implement some important algorithms imperatively;

Part two. To learn:

- the basics of modularising programs;
- how to specify abstract datatypes;
- how to implement some important data structures;
- how to formalise the relationship between an abstract datatype and its implementation.

Scala

We will use Scala as our programming language, because:

- It's fun;
- It has a clean, concise but readable syntax;
- It is heavily influenced by Haskell, and so has most of the useful high-level features of Haskell;
- It's an object-oriented language, which facilitates building large programs;
- It has a well designed library with lots of useful features.

However, this isn't a course about Scala: it's the principles that matter, and those principles are language-independent.

Administration

Lectures (3 per week):

- Monday 11am, Weeks 1–7;
- Tuesday 11am, Weeks 1–7;
- Thursday 11am, Weeks 1–6.

Problem sheets: 3 + 3.

- Part 1 sheets designed for tutorials in Weeks 2, 3, 4.
- Part 2 sheets designed for tutorials in Weeks 5, 6, 7.
- (Subject to arrangement with your tutor!)

Practicals:

- First practical (for Part 1), Weeks 2, 3, 4;
- Second practical (for Part 2), Weeks 6, 7, 8.

Reading

Main text: Programming in Scala, Odersky, Spoon and Venners, Chapters 1–7.

Subsidiary reading: Programming Pearls, Bentley.

A first example: factorial

Let's start with a very simple example, namely a function `fact` to calculate factorials. The first program does this in a functional style.

```
/** Calculate factorial of n
  * Pre: n >= 0
  * Post: returns n! */
def fact(n: Int) : Int = {
  if(n==0) 1
  else fact(n-1)*n
}
```

Syntactic notes

- Definitions of functions are introduced with the keyword `def`;
- The body of the function is included inside the curly brackets `{...}` (in fact, they're not necessary here);
- Types in Scala are given in Pascal style (with a single colon);
- Most typing information in Scala is optional, but types are necessary for arguments of functions and the return types of recursive functions; it's a good idea to give types elsewhere, when they help document the code;
- A "`return`" statement is optional so we've left it out;
- In `if` statements, the condition must be inside parentheses, and there is no "`then`".

Functions in brief

- Keyword `def`
- Last executed expression determines the returned value
- Result type may be inferred by Scala
- Keyword `Unit`
- Variables declared inside a function are local
- Functions can be used to decompose more complicated problems
- Syntax

```
def name(...):T = { ...  }
```

Conditionals in brief

- Keyword `if`, `else`, and `Boolean`
- Boolean expressions in Scala: true, false, etc.
- Equality test `==` vs. assignment `=`
- Comparisons: `==` vs. `eq` (won't be important until Part 3)
- Syntax

```
if (...)  { ...  } else { ...  }
```

- Also `match` and `case` (cases can be patterns):

```
x match {
  case 0 => "zero"
  case 1 => "one"
...
}
```

Loops in brief

- Keywords `while` and `for`
- Syntax

```
while(...) { ... }
```

- Syntax

```
for (...) { ... }
```

- Example of `<-` notation

```
for (i <- 0 to 4) print(i);  for (i <- 0 until 4) print(i);
```

- Note that in Part 1 we concentrate on `while` (to help with reasoning). Use of `for` is outlawed for the time-being.

The REPL interpreter

We can experiment with functions written in Scala using the REPL interpreter (invoked by typing `scala` at the command line).

An interactive interface similar to `ghci`, `hugs`, `ipython`, `clisp`....

("REPL" stands for "Read-Eval-Print Loop".)

Fixing a couple of bugs

The previous definition of `fact` doesn't work if `n` < 0; we'll assume `n` ≥ 0 from now on as a precondition. We can add the line

```
  require(n>=0)
```

to check this.

The previous definition of `fact` also doesn't work correctly with arguments greater than 12.

It's better if we arrange for `fact` to return a `BigInt`:

```
/** Calculate factorial of n
  * Pre: n >= 0
  * Post: returns n! */
def fact(n: Int) : BigInt = {
  require(n>=0)
  if(n==0) 1 else fact(n-1)*n
}
```

A larger program

We now want to incorporate the function `fact` within a larger program. When we run the program, it should behave as below:

```
Please input a number: 5
The factorial of 5 is 120
```

A larger program

```
object Factorial{
  /** Calculate factorial of n
    * Pre: n >= 0
    * Post: returns n! */
  def fact(n: Int) : BigInt = {
    require(n>=0)
    if(n==0) 1 else fact(n-1)*n
  }

  // Main method
  def main(args: Array[String]) = {
    print("Please input a number: ")
    val n = scala.io.StdIn.readInt
    val f = fact(n)
    println("The factorial of "+n+" is "+f)
  }
}
```

Syntactic notes

- An object encapsulates some code; typically an object will encapsulate some data and operations on that data (more later).
- Normally an object is defined in a file with a corresponding name, e.g. `Factorial.scala`.
- Each main object should have a function with header

```
def main(args: Array[String])
```

  (I'll explain this more later.)
- Values (which cannot be changed) are introduced with `val`.
- `print`, `println` and `readInt` are library functions.
- Here `+` represents string concatenation.
- Note the use of indentation to visually indicate the nesting.

Dealing with negative inputs

The previous code goes wrong if a negative number is input: it calls `fact` with an argument that doesn't meet its precondition.

The following version of `main` is better.

```
def main(args : Array[String]) = {
  print("Please input a number: ")
  val n = scala.io.StdIn.readInt
  if(n>=0){
    val f = fact(n)
    println("The factorial of "+n+" is "+f)
  }
  else println("Sorry, negative numbers aren't allowed")
}
```

Note that the curly brackets in the `if` clause are necessary to indicate that the clause consists of both statements.

Compiling and running programs

Programs can be compiled using `scalac`, e.g.

```
> scalac Factorial.scala
```

or better using `fsc`, e.g.

```
> fsc Factorial.scala
```

This will produce a corresponding `.class` file, e.g. `Factorial.class`, and (normally) some other `.class` files, containing Java bytecode.
(If `fsc` gets confused, try `fsc -shutdown`.)

The `.class` file can be executed on `scala`, e.g.

```
> scala Factorial
```

(This executes the bytecode on the Java Virtual Machine[a].)

[a] See `http://en.wikipedia.org/wiki/Java_virtual_machine`.

Summary

- Introduction to Scala;
- Functions, variables, `while` loops, ..., and lots of other pieces of syntax;
- Mechanics;
- Preconditions.
- Next time: Removing recursion; Invariants.

Reading: Read Chapters 1 and 2 of Programming in Scala. Also read Chapter 5 (excluding Section 5.6) within the next couple of weeks.