Lecture 6

Over the next two lectures, we'll look under the hood and discuss some of the ideas used to build the Scala compiler and the Java virtual machine. Most of the time, you will not have to think about these kinds of low-level details. However, to write performance-critical code or to debug low-level errors, you will need to know this kind of work.

Programs as Trees

These are examples of arithmetic expressions:

- `2 * 4`
- `1 + 2 + 3`
- `5 * 4 * 2`
- `1 + 2 * 3`

We all know how to evaluate these expressions in our heads. But, when we do, we resolve several ambiguities. For example, should we evaluate `1 + 2 * 3` like this:

```
    1 + 2 * 3
=   (1 + 2) * 3
=   3 * 3
=   9
```

or like this:

```
    1 + 2 * 3
=   1 + (2 * 3)
=   1 + 6
=   7
```

The latter is the convention in mathematics, but it is an arbitrary choice. A programming language could adopt either convention or adopt a completely different notation. For example, the following three programs, written in three different languages, are trying to express the same thing:

- `1 + 2` infix syntax, in Scala, which (mostly) adopts the notation of conventional mathematics
- `(+ 1 2)` parenthesized prefix syntax, from the Scheme language
- `1 2 +` postfix syntax, from the Forth language

These three *concrete syntaxes* are very different, but all mean "the sum of the number three and the number four".

Concrete syntax is important, because it is the human-computer interface to a programming language. It is easy to find acrimonious debates on the Web about the virtues of Python's indentation-sensitive syntax, whether semicolons should be optional in JavaScript, how C code should be indented, and so on. However, we can think of expressions more abstractly as *abstract syntax trees*.

Scala's case classes make it easy to define a type for abstract syntax trees of arithmetic expressions:

```
sealed trait Expr
case class Num(n: Int) extends Expr
case class Add(e1: Expr, e2: Expr) extends Expr
case class Sub(e1: Expr, e2: Expr) extends Expr
case class Mul(e1: Expr, e2: Expr) extends Expr
```

```
scala> import scala.tools.reflect.ToolBox
import scala.tools.reflect.ToolBox

scala> import scala.reflect.runtime.universe.runtimeMirror
   def runtimeMirror(cl: ClassLoader): reflect.runtime.universe.Mirror

scala> val tb = runtimeMirror(getClass.getClassLoader).mkToolBox()
tb: scala.tools.reflect.ToolBox[reflect.runtime.universe.type] = scala.tools.reflect.ToolBoxFactory$
    ToolBoxImpl@4bef0fe3

scala> val tree = tb.parse("""(x: Int, y: Int, z: Int) => x + y * z""")
tree: tb.u.Tree = ((x: Int, y: Int, z: Int) => x.$plus(y.$times(z)))

scala> tb.eval(tree)
res0: Any = <function3>

scala> res0.asInstanceOf[(Int, Int, Int) => Int](1, 2, 3)
res2: Int = 7
```

Figure 11.1: Printing a Scala abstract syntax tree.

Here are some examples of abstract arithmetic expressions and their concrete representations (written in normal, mathematical notation):

Concrete Syntax	Abstract Syntax
`1 + 2 + 3`	`Add (Add (Num(1), Num(2)), Num(3))`
`1 + 2 * 3`	`Add (Num(1), Mul (Num(2), Num(3)))`
`(1 + 2) / 3`	`Div (Add (Num(1), Num(2)), Num(3))`

The Scala compiler uses a much larger and more complex type to represent Scala abstract syntax trees, but it follows the same principle described here. You can use some low-level Scala APIs to take a string and turn it into an abstract syntax tree and even evaluate the tree, as shown in fig. 11.1.

Back to our arithmetic expression trees, we can write a simple recursive function evaluate trees to values as follows:

```
def eval(e: Expr): Int = e match {
  case Num(n) => n
  case Add(e1, e2) => eval(e1) + eval(e2)
  case Sub(e1, e2) => eval(e1) - eval(e2)
  case Mul(e1, e2) => eval(e1) * eval(e2)
  case Div(e1, e2) => eval(e1) / eval(e2)
}
```

You could evaluate Scala programs by writing a recursive evaluation function too. However, that's not what happens on your machines. To understand how Scala is run, it will help to first examine how Java programs are run.

Stacks of Instructions

Suppose you want to run the Java program in fig. 11.2. You can't simply run it directly. Instead, you need to first *compile* it:

```
javac Simple.java
```

which produces a file called `Simple.class`. You can run this file by starting the *Java Virtual Machine* (JVM):

```
java Simple
```

The latter command only uses `Simple.class`. You can delete `Simple.java` and you'll still be able to run the program. Let's look into what the `javac` and `java` commands do, before we return to Scala. Even if you've never used `java` and `javac` directly, and you've only ever used Java within an IDE such as Eclipse or IDEA, you should know that these IDEs run these commands under the hood.

The command `javac` is a *compiler* that translate Java source code to *byte-code*, which is stored in `.class` files. Byte-code is not designed to be human-readable, which is why `.class` are in a binary format that will appear are gibberish in a text editor. However, you can *disassemble* the byte-code and print it in a more human-readable format, by running `javap -c Simple.class`. This command produces the output shown in fig. 11.3.

Observe that all variables in the original program have been erased. Instead of using variables, Java byte-code uses a stack of values and a list of instructions, where the instructions pop and push the values on the stack.

```
class Simple {

  static int factorial(int n, int result) {
    if (n == 0) {
      return result;
    }
    else {
      return factorial(n - 1, result * n);
    }
  }

  public static void main(String[] args) {
    System.out.println("Hello, world!");
     System.out.println(factorial(5, 1));
  }

}
```

Figure 11.2: Java source code.

```
class Simple {
  Simple();
    Code:
       0: aload_0
       1: invokespecial #1
       4: return

  static int factorial(int, int);
    Code:
       0: iload_0
       1: ifne          6
       4: iload_1
       5: ireturn
       6: iload_0
       7: iconst_1
       8: isub
       9: iload_1
      10: iload_0
      11: imul
      12: invokestatic  #2
      15: ireturn

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #3
       3: ldc           #4
       5: invokevirtual #5
       8: getstatic     #3
      11: iconst_5
      12: iconst_1
      13: invokestatic  #2
      16: invokevirtual #6
      19: return
}
```

Figure 11.3: Java bytecode.

Figure 11.4: Disassembling Java.

```
sealed trait Instruction
case class INum(n: Int) extends Instruction
case class IAdd() extends Instruction
case class IMul() extends Instruction
case class ISub() extends Instruction

def eval(code: List[Instruction], stack: List[Int]): List[Int] = (e, stack) match {
  case (INum(n), stack) => n :: stack
  case (IAdd(), n1 :: n2 :: stack) => n1 + n2 :: stack
  case (ISub(), n1 :: n2 :: stack) => n1 - n2 :: stack
  ...
}
```

Figure 11.5: A stack-based evaluator for a list of arithmetic instructions.

```
def compile(e: Expr): List[Instruction] = e match {
  case Num(n) => List(INum(n))
  case Add(e1, e2) => compile(e1) ++ compile(e2) ++ List(IAdd())
  case Sub(e1, e2) => compile(e1) ++ compile(e2) ++ List(ISub())
  case Mul(e1, e2) => compile(e1) ++ compile(e2) ++ List(IMul())
}
```

Figure 11.6: A compiler from arithmetic expressions to arithmetic instructions.

Returning to arithmetic expressions, we can write a stack machine for arithmetic instructions as shown in **??**. The command `java Simple` runs a (very sophisticated) stack machine that is based on the same basic principle illustrated in this figure.

The following table shows some arithmetic expressions and equivalent instruction sequences

Concrete Syntax	Abstract Syntax	Instructions
`1 + 2 + 3`	`Add (Add (Num(1), Num(2)), Num(3))`	`List(INum(1), INum(2), IAdd(), INum(3), IAdd())`
`1 + 2 * 3`	`Add (Num(1), Mul (Num(2), Num(3)))`	`List(INum(1), INum(2), INum(3), IMul(), IAdd())`
`(1 + 2) / 3`	`Div (Add (Num(1), Num(2)), Num(3))`	`List(INum(1), INum(2), IAdd(), INum(3), IDiv())`

The command `javac Simple.java` performs the kinds of translations shown above automatically. The code in fig. 11.6 compiles arithmetic expressions to instructions using the same principles that `javac` uses.

The Scala compiler translates Scala source code to byte-code for the Java Virtual Machine in a similar way. The compiler is more complicated, because Scala is actually much more complex than Java. However, once the `.class` files have been generated, Scala programs run in the same manner at Java programs.

Although we use `sbt` in this course, it is also a simple frontend for the Scala compiler (`scalac`). You can run `scalac` on Scala source code to produce `.class` files and then disassemble or run these files in exactly the same way as Java code. In your `sbt` projects, there is a directory called `target/scala-2.11/classes` that has all the class files generated from your program. `sbt`, like any other IDE, puts `.class` files in a separate directory so that they don't clutter your code.