46 Syntax

Scala programs and syntax

- I’ve explained before about the nature of Scala programs:
    - Basically, a Scala program is an expression that yields a result* but…
        - there are other constructs which you can insert before the expression such as:
            - type definitions (traits, classes, etc.) (including methods and initialization—expressions yielding Unit);
            - method definitions (where the RHS is an expression);
            - val/var definitions (where the RHS is an expression);
            - imports;
            - syntactic sugar such as for-comprehension, case clause.
    - Usually, if you add such things to your expression you will need to create a block with {}

* So, you don’t need to end an expression with “return” since that’s what’s expected.

Lines and blocks

- Lines:
  - Scala lines don’t normally need a semi-colon at the end of each line because they are not normally *statements* (but there are exceptions to this which we will cover later—and which the compiler will warn you about)
  - The compiler pseudo-inserts a semi-colon for you at the end of each line if it thinks it’s a statement rather than an expression.
  - So, if you write:

    ```
    expression1
    +expression2
    ```

    it will be interpreted as a statement followed by an expression

    `expression1;` and `+expression2`
- You can fix this by putting parentheses around your expression or by moving the operator up to the first line:

  ```
  (expression1
  +expression2)
  ```

  ```
  expression1+
  expression2
  ```

Lines and blocks (contd.)

- Blocks:
  - they allow you to precede your expression by val/var/def statements;
  - they reduce namespace conflicts;
  - they allow for encapsulation (information hiding);
  - there is no requirement for method or identifier definitions (defs, vals, etc.) to be at any particular level: private methods will normally be inside a public method definition (or a class)
  - Even import statements can be inside blocks

```
val x = 3
def y = {
  val z = sqr(y)*x
  import math.round
  def sqr(p: Double) = round(p*p).toInt
}
```

Basic Scala syntax*

- module ::= prolog type-definition*
- prolog ::= package import*
- type-definition ::= header definition* “}”
- header ::= trait identifier mixins “{“ |
  [“abstract”|”case”] “class” identifier type-declaration parameter-set* mixins “{“ auxiliary-constructor* initialization |
  object identifier “{“
- mixins ::= “extends” type [“with” type”]*
- definition ::= method-definition | variable-definition | type-definition
- method-definition ::= “def” identifier [parameter-set]* “:” return-type [“=“ expression]
- variable-definition ::= [“lazy”] “val”|”var” identifier “:” return-type [“=“ expression]
- expression ::= identifier | invocation | “{“ definition* expression “}”
- invocation ::= [receiver] [“.”] identifier [identifier | [“(“ expression* “)”]*

* For the true syntax see http://www.scala-lang.org/docu/files/ScalaReference.pdf p. 159

Parentheses

- Parentheses are generally there to override the precedence rules for expressions. But occasionally, there's a bit more to it.
  - IntelliJ/IDEA and Eclipse have analyzers which will tell you if you have superfluous parentheses.
    - Joke: *what does LISP stand for?* Lots of irritating superfluous parentheses
  - For example, you don't need parentheses around a singleton parameter type of a function type—these are the same:

```
val x_f_t: Try[(X) => Fitness] = for ((t, s) <- matchFactors(factor, `trait`)) yield fc(t)(s)
val x_f_t: Try[X => Fitness] = for ((t, s) <- matchFactors(factor, `trait`)) yield fc(t)(s)
```

  - But, it's fairly conventional to put the parentheses there, in parallel so to speak with the function invocation. And if your parameter list is a tuple ("parameter set"), you must use parentheses.
  - You don't need parentheses for a lambda:

```
for (x <- RNG.values(r)) yield modulo(x, _ % mnopc)
```

Patterns—Review (1)

- In Scala, pattern-matching plays a big part. Patterns are found:
  - in a case clause (within a match);

    ```
    case Some(x) => println(x)
    ```

  - in a lambda;

    ```
    map (x => 2*x)
    map {x: Int => 2*x}
    ```

  - in a for-comprehension.

    ```
    for (x <- xs) yield x*2
    for (Some(x) <- xos) yield x*2
    ```

    - *BTW, some of these are very subtly similar (I don't even understand some of the distinctions—I use the source-code analyzer to help in some situations)*
- The important thing is that a pattern not only *matches* but also serves as a pseudo-variable within its scope.

Patterns (2)

- Example:

  ```
  def map[U](f: (T) => U): RandomState[U] =
  JavaRandomState[U](n, n => f(g(n)))
  ```

  - In this fragment of code, there are two “n”s. The first *n* is a variable in the scope of the *map* method and its enclosing class. The second *n* is a pattern (within a lambda). It could equally have been *x* (probably should have). The lambda could also have been written (with no explicit pattern):

    ```
    f(g(_))
    ```

- Another form that is basically the same:

  ```
  def map[U](f: (T) => U): RandomState[U] =
  JavaRandomState[U](n, {m:Long => f(g(m))})
  ```

Patterns (3) and “_”

- The match-everything pattern: _
  - Similar to a simple identifier like x, which also matches everything, the “_” does not define a bound variable, e.g. *case _ => None*
- But the underscore _ has several other meanings:
  - Anonymous bound variable in a lambda, e.g. _+_
  - The wildcard in an import statement (like ‘*’ in Java)
  - Higher-kinded type parameter, e.g. *def f[M[_]]*
  - **η**-expansion of method into function, e.g. *apply _*
  - conversion of sequence to varargs: as in *f(xs: _*)* or as in *case Seq(xs @ _*)*