Syntactic Sugar and Pattern Matching

Syntactic Sugar

- Scala has a neat feature called "syntactic sugar" which:
  - allows us to write a program in a very human-readable form that gets converted into more computer-readable form;
  - perhaps the most obvious example is the for-comprehension:

```
for(x <- List(1,2,3)) yield x*x
```

=> (i.e. is de-sugared into)

```
List(1,2,3).map {x => x*x}
```

and

```
for(x <- List(1,2,3); y <- List(4,5,6)) yield x*y
```

=>

```
List(1,2,3).flatMap { x => List(4,5,6).map { y => x*y } }
```

Syntactic Sugar (2)

- Desugaring continued…
  - other examples:

```
val x = 1; x + x
```

=> (i.e. is de-sugared into)

```
val x: Int = 1; x.+(x)
```

and

```
class X[Y : Numeric] {}
```

yields a constructor of the form

```
def <init>()(implicit ev$1: Numeric[Y]): this.X[Y] = {X.super.<init>();()}
```

and

```
List(1,2,3).map(_*2)
```

=>

```
val x = List(1, 2, 3).map(((x$1) => x$1.$times(2)))
```

Syntactic Sugar (3)

- Unary functions (i.e. instances of *Function1*):
  - `f(x) → f.apply(x)`
  - most collection-type containers implement *Function1,* e.g. *List, Map, Set* but not *Option, Try, Either*.
- A unary method on an object (like *map*, for example) is a binary (dyadic) function on the object and the parameter:
  - `a map f → a.map(f)`

Note that *map* associates to the left: in this case, *a* is the receiver; *f* is the parameter

- But, a method/operator whose name ends in ":", associates to the *right*:
  - `h :: t → t.::(h)`
- And, for mutable objects, there is a built-in assignment method:
  - `x(y) = z → x.update(y,z)`
- And, obviously, tuples:
  - `(x,y) → Tuple2[X,Y](x,y)`
  - `(x,y,z) → Tuple3[X,Y,Z](x,y,z)`

Syntactic sugar (4)

- And another example of our old friend the for-comprehension (How does yield work):

```
for { i <- List(1,2,3); x = i*3; if (x%2 == 0)} yield x  →
    List(1,2,3) map (_ * 3) filter (_ %2 == 0)
```

```
for(x <- c1; y <- c2; z <- c3) yield {…}  →
  c1.flatMap(x => c2.flatMap(y => c3.map(z => {...})))
```

Syntactic Sugar (5)

- It's important to realize that there is **no magic** in syntactic sugar (or any other aspect of Scala).
  - The desugaring process (the first process in the compiler's workflow) is a formal substitution scheme just as you'd expect
  - IntelliJ/IDEA offers a desugaring option;
  - You can see the result of desugaring a module by (e.g.):

    ```
    scalac -Xprint:parser src/main/scala/edu/neu/coe/scala/FutureExercise.scala
    ```

  - You can see other options from the compiler by using:

    ```
    scalac -Xshow-phases
    ```

  - You can see, for example, the "trees" for an expression:

    ```
    scala -Xprint:typer -e "val x = Option(1); x match { case Some(y) =>
    println(y); case _ => }"
    ```

Other Scala constructs

- call-by-value and call-by-name:

```
def f(x: X)
def f(x: => X)
```

"=> X" is a nullary function that results in an *X*. Can also be written "() => X"

- anonymous functions:

```
List(1,2,3) (_.toString)
List(1,2,3) (_*_)
List(1,2,3) (x =>x*x)
```

"_" stands for the obvious: a particular element of *m*

No. Doesn't work with *List(Int)* but would work with *List((Int,Int))* for instance.

This is OK as we have explicitly named the value

- varargs methods:

```
def sum(xs: Int*) = xs reduce (_+_)
sum(List(1,2,3): _*)
```

"*" tells the compiler that the parameter args is a variable sequence of *Int*, not just one Int.

- tuples: defining

```
scala> val x = 1->"a"
x: (Int, String) = (1,a)
```

For example, initializing a *Map*:
`val x = Map(1->"a", 2->"b",…)`

Standard Imports

- When looking for operators, methods, implicit functions, values, it makes sense to know what's automatically imported:

```
import java.lang._    // http://docs.oracle.com/javase/8/docs/api/java/lang/package-summary.html
import scala._        // http://www.scala-lang.org/api/current/#scala.package
import scala.Predef._ // http://www.scala-lang.org/api/current/#scala.Predef$
```

Pattern Matching (review)

- First: expressions:
  - The most basic aspect of functional programming is that it allows you to define expressions which yield some result.
  - You can split a complicated expression up using *val* or *def* and make things easier to understand (and usually shorter):
  - Let's say we want to know the sum of the integers 1 thru 20:

```
scala> 1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17+18+19+20
res0: Int = 210

scala> 20*(20+1)/2
res1: Int = 210

scala> val n = 20
n: Int = 20

scala> n*(n+1)/2
res2: Int = 210

scala> def sumOfNIntegers(n: Int) = n*(n+1)/2
sumOfNIntegers: (n: Int)Int

scala> sumOfNIntegers(20)
res3: Int = 210
```

  - We've gone from the most specific form to the most general form. All give the same answer: 210. That's because all expressions are mathematically identical. The final answer is easiest to read, however. The first answer could easily be in error and we humans might not notice (a number skipped or repeated, for example).

Pattern Matching (2)

- There's another way we could have defined the sum…
  - Suppose we create a *case class* to represent a range of numbers starting with 1:

```
scala> case class Range(n: Int) { def sum = n*(n+1)/2 }
defined class Range

scala> val r = Range(20)
r: range = Range(20)

scala> r sum
res4: Int = 210
```

  - We've talked about extractors and pattern matching before. Essentially, they are the opposite of the substitution principle that we use in expressions. Extractors are like a "what if?" scenario:

```
scala> r match { case Range(m) => println(m) }
20
```

  - **What if** we had a *Range* such that its *n* value was represented by a variable called *m*? We could print *m* to see what it was or do anything else with *m*.
  - It's important to understand that *m* is a "variable" (in the algebraic sense) but it's value is fixed by the pattern-matching code to be whatever *n* was in the *Range*.