Updated: 2021-02-18

3.11
Pattern Matching
The ”dark” side of expressions

Northeastern University

A deeper dive into Pattern Matching

- We looked at pattern matching in 2.3 Functional Programming in Scala.
- Let’s do a “deeper dive” into the subject and see how expressions and pattern-matching are really two sides to the same coin.

Expressions

- Expressions
  - One of the central concepts in almost all languages is the *expression.*
  - An expression is essentially a function which takes some number of variables/constants/expressions and yields a result, e.g. *f(x,y,z)* or *x+z*2.* Remember that (in Scala) this is just a human-readable form of *x.+(z.*(2))*.
  - In procedural languages, expressions form the right-hand-sides of statements such as assignments, return statements, constant declarations, the branches of an if/else/for/while/do construct, etc.
  - In Scala, the yield (return value) of *every part of the language is an expression*. The only exceptions are:
    - declarations of traits/classes/objects, methods and variables;
    - other non-expression constructs: import, package, “type” statements.

Patterns

- *Patterns* are the antithesis (i.e. opposite) of expressions and are found only in functional programming languages.
- A pattern allows you to take a variable and *produce* several variables (exactly the opposite of what an expression does).
- Let’s look at an example:

```scala
def decode(x: (Int,String)): Unit =
 x match {
    case (k,v) => println(s"$k: $v")
  }
```

- In this example, we are pattern-matching on a *Tuple* of *Int* and *String*. We only need one case whose pattern looks exactly like an expression forming a tuple from *k* and *v*. But it is a pattern instead and so *k* and *v* are extracted from the given tuple *x*, and become in-scope variables which can be used in the expression on the right-hand side, in this case *println*…
- Thus, a pattern which looks like an expression is actually an *extractor.*

Patterns (1)

- Let’s look at another example:

```
def print(xs: List[Int]): Unit =
    xs match {
      case h :: t => println(h); print(t)
      case _ =>
     }
```

- In this example, the construct $h :: t$ is a pattern. It looks *exactly* like an expression BUT instead of *h* and *t* being existing in-scope variables which will be applied to the *::* operator to yield a value, instead the value is matched against the pattern and, if there is a match, variables *h* and *t* are, magically if you like, declared and in scope.
- In this situation, it is the variable *xs* which is matched against the patterns—in the order defined—and, if a match is found, the expression on the RHS of the “=>” (“rocket symbol”) is invoked. Here, *h* is passed to *println* and *t* is passed, recursively, to *print.*

Patterns (2)

- Here again is our example:

```
def print(xs: List[Int]): Unit =
    xs match {
     case h :: t => println(h); print(t)
     case _ =>
     }
```

- There is also a catch-all pattern "_", although any identifier name would do, such as "a", "x", etc. The unique thing about the "_" is that the pattern matches as it would to an identifier, but no variable is synthesized.
- I need to explain what $h :: t$ actually represents (whether as an expression or a pattern).
  - In the context of an expression, "::" is actually the case class "::" (which we pronounce "cons") or, more precisely, the auto-magically generated *apply* method of the companion object "::"

Patterns (3)

- Here again is our example (but slightly altered):

```
def copy(xs: List[Int]): List[Int] =
   xs match {
      case h :: t => h :: t
      case Nil => Nil
   }
```

- When we write $h :: t$ as an expression the compiler actually expands this into $::.apply(h, t)$.
- But when we write $h :: t$ as a pattern, a completely different mechanism is used. The compiler invokes the following method on "::":
  - *unapply[X](xs: List[X]): Option[(X,List[X])]* which results in an (optional) tuple of two parameters: an *X* (the head) and a *List[X]* (the tail). Any object can have an *unapply* method, but the magic of case classes is that they auto-generate the appropriate *unapply* method. That's to say that case classes are *designed* for use in pattern-matching!
- Can you see that $h :: t$ as an expression and $h :: t$ as a pattern are simply two sides to the same coin? One expands to $::.apply(h, t)$ and the other, effectively, to $val\ (h,t) = ::.unapply(xs).get$

Patterns (4)

- Here again is our example (but altered again):

```
def reverse(xs: List[Int]): List[Int] =
   xs match {
      case h :: t => reverse(t) :+ h
      case Nil => Nil
   }
```

- A **case class** is just like an ordinary class but it has some very special additional properties. Our particular case class is defined in the API thus:

  final case class **::**[B](head: B, tl: List[B]) extends List[B] with Product with Serializable

- The additional properties that the compiler provides for a case class are:
  - the members of the case class are, by default, available as methods of the class (e.g. *xs.head*); the case class instances also behave as *Tuples* (as provided for by the *Product* trait);
  - *toString*, *equals* and *hashCode* methods are all automagic;
  - the companion object provides an *apply* method (which effectively obviates the need to use the "new" keyword) and an *unapply* method which, given an instance of the case class, will yield an optional tuple of the parameters. **This** is how the variables *h* and *t* are automagically generated from the $h :: t$ pattern.

Patterns (5)

- Back to patterns. What other sorts of patterns are there?
  - constant patterns, for example using *Nil* (the empty list):

```
xs match {
  case Nil =>
    case h :: t => println(h); print(t)
  }
```

  - typed patterns, for example (the parentheses are not always required) where we are only interested in *Int* values:

```
xs match {
  case Nil =>
    case (h: Int) :: t => println(h); print(t)
    case _ =>
  }
```

  - arbitrarily-nested patterns, for example:

```
xs match {
  case Nil =>
    case h :: k :: t => println(h+k); print(t)
    case _ =>
  }
```

  - guarded patterns, for example:

```
xs match {
 case Nil =>
   case h :: t if h>0 => println(h); print(t)
 case _ => println("head was not positive")
}
```

Where can we use patterns?

- The most obvious context for a pattern is after the *case* keyword in a *match* clause.
- But, don’t forget about patterns after **val** or before “<-” in a *for-comprehension*:

```
case class Complex(real: Double, imag: Double)
val pi = Complex(-1, 0)
val Complex(r, i) = pi
for (x <- xs) println(x)
```

- Do you see the elegant symmetry of the two middle lines? In particular, note that:
  - Every value on the *right* of the “=“ must evaluate to something;
  - Every value on the *left* of the “=“ actually defines a “bound variable”.

Wrap-up

- We will come across patterns again when we talk about for comprehensions. Note that you can see the details of my examples in running code in the REPO. Just look for *PatternExample*.