Programming Paradigms Tutorials
Pattern matching

Pattern matching

Pattern matching is the second most widely used feature of Scala, after function values and closures. Scala provides great support for pattern matching, in processing the messages.

A pattern match includes a sequence of alternatives, each starting with the keyword case. Each alternative includes a pattern and one or more expressions, which will be evaluated if the pattern matches. An arrow symbol => separates the pattern from the expressions. Example:

```
def matchTest(x: Int): String = x match {
    case 1 => "one"
    case 2 => "two"
    case _ => "many"
}
```

The following method shows examples of many different types of patterns you can use in match expressions:

```
def echoWhatYouGaveMe(x: Any): String = x match {

    // constant patterns
    case 0 => "zero"
    case true => "true"
    case "hello" => "you said 'hello'"
    case Nil => "an empty List"

    // sequence patterns
    case List(0, _, _) =>
     "a three-element list with 0 as the first element"
    case List(1, _*) =>
     "a list beginning with 1, having any number of elements"
    case Vector(1, _*) =>
     "a vector starting with 1, having any number of elements"

    // tuples
    case (a, b) => s"got $a and $b"
    case (a, b, c) => s"got $a, $b, and $c"

    // constructor patterns
    case Person(first, "Szczepanik") =>
     s"found an Szczepanik, first name = $first"
    case Dog("Lesse") => "found a dog named Lesse"

    // typed patterns
```

```
    case s: String => s"you gave me this string: $s"
    case i: Int => s"thanks for the int: $i"
    case f: Float => s"thanks for the float: $f"
    case a: Array[Int] => s"an array of int:
     ${a.mkString(",")}"
    case as: Array[String] =>
     s"an array of strings: ${as.mkString(",")}"
    case d: Dog => s"dog: ${d.name}"
    case list: List[_] => s"thanks for the List: $list"
    case m: Map[_, _] => m.toString

    // the default wildcard pattern
    case _ => "Unknown"
}
```

Algebraic Data Types

Algebraic Data Types (ADTs for short) are a way of structuring data. They're widely used in Scala due, mostly, to how well they work with pattern matching and how easy it is to use them to make illegal states impossible to represent.
There are two basic categories of ADTs:

- product types
- sum types

Product type

A product type is essentially a way of sticking multiple values inside of one - a Tuple, or something that's very similar to one. Case classes are the prototypical product type:

```
final case class Foo(b1: Boolean, b2: Boolean)
```

Foo aggregates two Boolean values.
It's called a product type because we can compute its arity (the number of values it can possibly have) by calculating the product of the types that compose it.

Sum types

A sum type is a type that is composed of different possible values and value shapes. The simplest possible example is an enumeration - Bool, for example:

```
sealed abstract class Bool extends Product with Serializable

object Bool {
  final case object True extends Bool
  final case object False extends Bool
}
```

It's called a sum type because its arity is equal to the sum of the arities of the types that compose it. Here, both True and False are singleton types, and Bool can indeed only have 2 possible values.

Exercises

1. (3pt.) Write a function which return all primes from list of numbers (limit values to: 0-200). Hint: use Sieve of Eratosthenes to generate list of primes and use filter for the list.

2. (3pt.) Using ADT and pattern matching, write a simple calculator that supports two functions: add (a + b) and negation (-n).

3. (2pt.) Using ADT define your own Bool type and write functions AND, OR, XOR, NAND, NOR.

4. (2pt.) Using pattern matching write function which print information what is the type or argument. It should support 5 types.