

CSC 347 - Concepts of Programming Languages

Pattern Matching

Instructor: Stefan Mitsch



Learning Objectives

- ❓ How to decompose and process complex nested data structures?
 - Identify matched expression and cases in Scala pattern matching
 - Express functions with pattern matching



Pattern Matching Example: Lists



1 In Scala, implement a method that takes a list of at least 3 numbers and returns it with the first 3 numbers sorted in ascending order



```
1 def f (numbers: List[Int]) : List[Int] = {  
2   require(numbers.length >= 3, "The list must contain at least 3 elements")  
3   val (firstThree, rest) = numbers.splitAt(3)  
4   firstThree.sorted ++ rest  
5 } ensuring { (result: List[Int]) => ??? }
```

- Contract for checking the first 3 elements

Index access

```
1 ensuring {  
2   (result: List[Int]) =>  
3     val x1 = result(0)  
4     val x2 = result(1)  
5     val x3 = result(2)  
6     x1 <= x2 && x2 <= x3  
7 }
```

Projections

```
1 ensuring {  
2   (result: List[Int]) =>  
3     val x1 = result.head  
4     val x2 = result.tail.head  
5     val x3 = result.tail.tail.head  
6     x1 <= x2 && x2 <= x3  
7 }
```

Pattern matching

```
1 ensuring {  
2   (result: List[Int]) =>  
3     val x1 :: x2 :: x3 :: _ = result  
4     x1 <= x2 && x2 <= x3  
5 }
```

```
1 ensuring {  
2   case x1 :: x2 :: x3 :: _ =>  
3     x1 <= x2 && x2 <= x3  
4 }
```



Pattern Matching on Tuples

💡 Pattern matching *branches* and *binds pattern variables*

⚡ Sum of the components of a tuple

- Decomposition with index access

```
1 def sum (p: (Int,Int)) : Int =  
2   if p==null then throw MatchError(p)  
3   val x = p(0)  
4   val y = p(1)  
5   x + y  
6 end sum
```

- Pattern matching

```
1 def sum (p: (Int,Int)) = p match  
2   case (x,y) => x+y
```

- With types (optional)

```
1 def sum (p: (Int,Int)) : Int = p match  
2   case (x: Int, y: Int) => x+y
```

- Every `case args => body` is a function



Pattern Matching on Lists

💡 Pattern matching *branches* and *binds* pattern variables

🔧 Print first element of a list

- Decomposition with projections

```
1 def printHead(xs: List[Int]) : String =  
2   if xs == Nil then "List is empty"  
3   else  
4     val y: Int = xs.head  
5     val ys = xs.tail  
6     s"List is non-empty, head is $y"  
7   end if  
8 end printHead
```

- Decomposition with pattern matching

```
1 def printHead(xs: List[Int]) : String = xs match  
2   case Nil =>  
3     "List is empty"  
4   case (y: Int) :: ys =>  
5     s"List is non-empty, head is $y"  
6 end printHead
```



Pattern Matching on Lists

- Omit unnecessary variables and types
- Decomposition with projections

```
1 def printHead(xs: List[Int]) =  
2   if xs == Nil then "List is empty"  
3   else  
4     val y = xs.head  
5     // val ys = xs.tail  
6     s"List is non-empty, head is $y"  
7 end printHead
```

- Decomposition with pattern matching

```
1 def printHead(xs: List[Int]) = xs match  
2   case Nil      => "List is empty"  
3   case y :: _   => s"List is non-empty, head is $y"  
4 end printHead
```

- Wildcard operator `_` means *don't care*



Nested Patterns

💡 Nested patterns: patterns can include other patterns

🔧 Print first tuple of a singleton list, else print second int

```
1 def f (xs: List[(Int,String)]) = xs match
2   case Nil                => "List is empty"
3   case x :: Nil           => s"List has one element: $x"
4   case _ :: (x,_) :: _    => s"The second int is $x"
5 end f
6
7 val zs = List ((1,"dog"), (2,"cat"), (3,"sloth"))
8 f(zs) // 2
```



Pattern Matching Exercise: List Operations

- Implement simple list operations by pattern matching

isEmpty

```
1 def isEmpty (xs: List[Int]) =  
2   xs match  
3     case Nil => true  
4     case _   => false  
5 end isEmpty
```

head

```
1 def head (xs: List[Int]) =  
2   xs match  
3     case Nil      =>  
4       throw NoSuchElementException()  
5     case y :: _ => y  
6 end head
```

tail

```
1 def tail (xs: List[Int]) =  
2   xs match  
3     case Nil      =>  
4       throw NoSuchElementException()  
5     case _ :: ys => ys  
6 end tail
```

- Many list operations are builtin:
 - List (1, 2, 3).head
 - List (1, 2, 3).tail
 - List (1, 2, 3).isEmpty



Summary

- Pattern matching to **decompose** lists, tuples, and objects into their components
- Pattern matching **branches and binds variables**
- Every `case args => body` is a function
- First matching function is evaluated