CSC 347 - Concepts of Programming Languages

Pattern Matching

Instructor: James Riely

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

```
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
```

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

- Contract for checking the first 3 elements

Index access

```scala
ensuring {
  (result: List[Int]) =>
    val x1 = result(0)
    val x2 = result(1)
    val x3 = result(2)
    x1 <= x2 && x2 <= x3
}
```

Projections

```scala
ensuring {
  (result: List[Int]) =>
    val x1 = result.head
    val x2 = result.tail.head
    val x3 = result.tail.tail.head
    x1 <= x2 && x2 <= x3
}
```

Pattern matching

```scala
ensuring {
  (result: List[Int]) =>
    val x1 :: x2 :: x3 :: _ = result
    x1 <= x2 && x2 <= x3
}
```

```scala
ensuring {
  case x1 :: x2 :: x3 :: _  =>
    x1 <= x2 && x2 <= x3
}
```

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