CSC 347 - Concepts of Programming Languages

Pattern Matching

Instructor: Stefan Mitsch

Learning Objectives

❓ How to decompose and process complex nested data structures?

- Understand pattern matching in Scala
- Revisit recursion in Scala

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)
  val sortedFirstThree = firstThree.sorted
  sortedFirstThree ++ rest
} ensuring { case xs => ??? }
```

- Index access

```scala
ensuring {
  case xs =>
    val x1 = xs(0)
    val x2 = xs(1)
    val x3 = xs(2)
    x1 <= x2 && x2 <= x3
}
```

- Projections

```scala
ensuring {
  case xs =>
    val x1 = xs.head
    val x2 = xs.tail.head
    val x3 = xs.tail.tail.head
    x1 <= x2 && x2 <= x3
}
```

- Pattern matching

```scala
ensuring {
  case xs =>
    val x1 :: x2 :: x3 :: Nil = xs
    x1 <= x2 && x2 <= x3
}
```

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

Pattern Matching

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

- 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
```

- 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
```

Pattern Matching on Lists

- Pattern matching branches and binds *pattern variables*
- 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"
```

- Decomposition with pattern matching

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

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"
```

- 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"
```

- Wildcard operator `_` means *don't care*
- Found in ML, Haskell, Rust, Swift, and coming to Java

Pattern Matching

- Nested patterns: patterns can include other patterns

```
1 def f (xs: List[(Int,String)]) = xs match
2   case Nil                => "List is empty"
3   case _ :: Nil           => "List has one element"
4   case _ :: (x,_) :: _ => s"The second int is ${x}"
5
6 val zs = List ((11,"dog"), (21,"cat"), (31,"pig"))
7 f(zs)
```

Pattern Matching

- Pattern matching vs. Projections
- Decomposition with pattern matching

```
1 def f (xs: List[(Int,String)]) = xs match
2   case Nil             => "List is empty"
3   case _ :: Nil        => "List has one element"
4   case _ :: (x,_) :: _ => s"The second int is ${x}"
5
6 val zs = List ((11,"dog"), (21,"cat"), (31,"pig"))
7 f(zs)
```

- Decomposition with projections

```
1 def f (xs: List[(Int,String)]) =
2   if xs == Nil then "List is empty"
3   else if xs.tail == Nil then "List has one element"
4   else s"The second int is ${xs.tail.head(0)}"
5
6 val zs = List ((11,"dog"), (21,"cat"), (31,"pig"))
7 f(zs)
```

Pattern Matching Exercise: List Operations

- Implement simple list operations by pattern matching

▶ `isEmpty`

▶ `head`

▶ `tail`

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

Recursion

- Imperative programming typically favors
  - mutable data
  - iteration using loops (`while` , `for`)
- Functional programming typically favors
  - immutable data
  - iteration using recursion
- Recursion requires efficient method calls
- State of computation
  - Imperative: loop counters to access "global" mutable data
  - Recursion: arguments to recursive call

Exercise: Length of List

- Imperative implementation

```
1 def length (xs:List[Int]) : Int =
2   var length: Int = 0
3   var current = xs
4   while current != Nil do
5     length = length + 1
6     current = current.tail
7   length
```

- Recursive with pattern matching

```
1 def length (xs: List[Int]) : Int = xs match
2   case Nil     => 0
3   case _ :: ys => 1 + length (ys)
```

- With parametric polymorphism

```
1 def length [X] (xs: List[X]) : Int = xs match
2   case Nil     => 0
3   case _ :: ys => 1 + length (ys)
```

Recursion

- Imperative iteration

```
1 length (List (1, 2, 3))
2 --> current = 1::(2::(3::Nil)), length = 0
3 --> current = 2::(3::Nil),      length = 1
4 --> current = 3::Nil,           length = 2
5 --> current = Nil,              length = 3
```

- The state of the computation is in mutable variables

- Recursive iteration

```
1 length (List (1, 2, 3))
2 --> length (1::(2::(3::Nil)))
3 --> 1 + length (2::(3::Nil))
4 --> 1 + (1 + length (3::Nil))
5 --> 1 + (1 + (1 + length (Nil)))
6 --> 1 + (1 + (1 + 0))
7 --> 1 + (1 + 1)
8 --> 1 + 2
9 --> 3
```

- The state of the computation is the expression

Appending Lists

```
1 def append [X] (xs: List[X], ys: List[X]) : List[X] = xs match
2   case Nil     => ys
3   case z :: zs => z :: append (zs, ys)
```

```
1 append (1::(2::Nil), 3::Nil)
2 --> 1::(append (2::Nil, 3::Nil))    // z = 1, zs = 2::Nil
3 --> 1::(2::(append (Nil, 3::Nil)))  // z = 2, zs = Nil
4 --> 1::(2::(3::Nil))                // z = 2, zs = Nil
```

- New elements `1` and `2` created
- List `3::Nil` is reused (shared)
- New list, but second part is shared!

Appending Lists

- `List` class has builtin method `:::`

```
1 scala> ((1 to 5).toList) ::: ((10 to 15).toList)
2 res1: List[Int] = List(1, 2, 3, 4, 5, 10, 11, 12, 13, 14, 15)
```

Recursion

- What does `f` do?

```
1 def f [X] (xs: List[X]) : List[X] = xs match
2   case Nil     => Nil
3   case y :: ys => f (ys) ::: List (y)
```

▶ `f (Nil)`

▶ `f (3::Nil)`

▶ `f (2::3::Nil)`

▶ `f (1::2::3::Nil)`

- Conclusion: `f` is `reverse`

Summary

- Pattern matching to decompose lists, tuples, and objects into their components
- Functional programming favors immutable data and recursion over mutable data and iteration