CSC 347 - Concepts of Programming Languages

Scala Introduction

Instructor: Stefan Mitsch

Learning Objectives

❓ How to combine multiple programming paradigms in a single language?

- Understand Scala basic syntax
- Understand functional programming in Scala
- Understand lists in Scala
- Revisit recursion in Scala

Scala

- Functional and object-oriented PL
- Java + ML + more
- Scalable Component Abstractions
- Compiles to JVM
- Interop: Scala calls Java; Java calls Scala
- Examples
  - Twitter/X Scala School
  - Apache Spark (Scala, Java, Python, R)
  - Chicago Scala Meetup

Scala

- Scala has a REPL like Scheme
- Boolean literals: `false || true`
- Numeric literals: `1 + 2`
- String literals: `("hello" + " " + "world").length`
- Use of Java's libraries

```
val dir = java.io.File ("/tmp")
dir.listFiles.filter (f => f.isDirectory && f.getName.startsWith ("c"))
```

- With explicit nulls enabled:

```
val dir = java.io.File ("/tmp")
dir.listFiles.nn.map(_.nn).filter (f => f.isDirectory && f.getName.nn.startsWith ("c"))
```

Everything is an Object

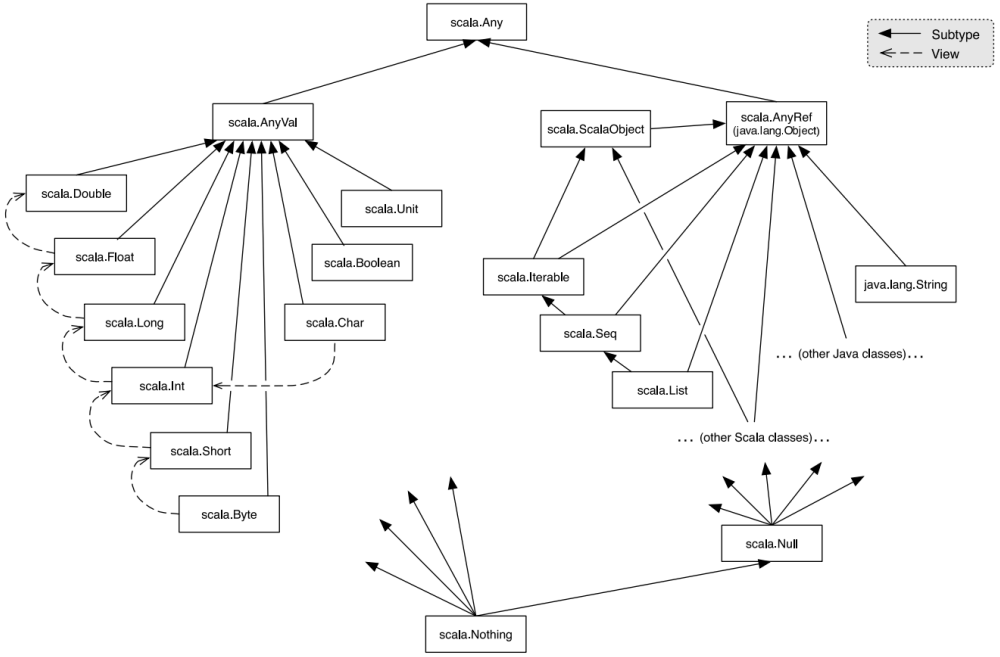
- `5:Int` is an object of type `Int` with methods: `5.toDouble`
- Methods can have symbolic names (see `scala.Int`): `5.+ (6)`
- `scala.runtime.RichInt` adds more methods: `5.max (6)`
- Any unary function `e1.f(e2)` can be written as `e1 f e2`
  - `5 + 6` is `5.+ (6)`
  - `5 max 6` is `5.max (6)`

Scala Type Checking

- Scala performs static type checking

```
def f () = 5 - "hello"   // rejected by type checker
```

- REPL prints types of expressions
- Java type hierarchy is embedded in Scala
  - Java primitive types are Scala value types
  - Java reference types are Scala reference types
  - `java.lang.Object` is `scala.AnyRef`



Mutable and Immutable Variables

Mutable Variables

- Java

```java
int x = 10;          // declare and initialize x
x = 11;              // assignment to x OK
```

- C

```c
int x = 10;          // declare and initialize x
x = 11;              // assignment to x OK
```

- Scala

```scala
var x = 10           // declare and initialize x
x = 11               // assignment to x OK
```

Immutable Variables

- Java

```java
final int x = 10;  // declare and initialize x
x = 11;            // assignment to x fails
// error: cannot assign a value to final variable x
```

- C

```c
const int x = 10;  // declare and initialize x
x = 11;            // assignment to x fails
// error: assignment of read-only variable ‘x’
```

- Scala

```scala
val x = 10           // declare and initialize x
x = 11               // assignment to x fails
// error: reassignment to val
```

Expression Sequencing

C Expressions

```
(e_1, e_2, ..., e_n)
```

C Statements

```
{
  s_1;
  s_2;
  ...
  s_n;
}
```

Scheme Expressions

```
(begin e_1 e_2 ... e_n)
```

Block-like format

```
(begin
  e_1
  e_2
  ...
  e_n
)
```

Scala Expressions

```
{e_1; e_2; ...; e_n}
```

Semicolons optional

```
{
  e_1
  e_2
  ...
  e_n
}
```

Methods

- Parameters require type annotations

```
def plus (x:Int, y:Int) : Int = x + y
def times (x: Int, y:Int)     = x * y
```

- Return types
  - can often be inferred
  - but are required for recursive methods
- Body of a method is an expression; its value is returned

Methods

- Conditional expressions

```
def fact (n:Int) : Int = if n <= 1 then 1 else n * fact (n - 1)
```

- Compound expressions for side-effects

```
def fact(n:Int) : Int =
  println("called with n=%d".format(n))
  if n <= 1 then
    println("no recursive call")
    1
  else
    println("making recursive call")
    n * fact(n - 1)
```

- Syntax like C statements, but are expressions!

Methods vs. Fields

- `def` can be used non-parameterized: `def x = 5`; non-strict, executed every time
- `val` declares a variable: `val x = 5`; strict, initialized once
- `lazy val`: memoized `def`, initialized on demand

Scala

```
class C:
  val x = 1
  lazy val y = 1 + 2
  def z = 1
```

Java

```
public class C {
  private final int x = 1;
  private Integer y = null;
  public int x() { return x; }
  public int y() {
    if (y == null) y = 1 + 2;
    return y;
  }
  public int z() { return 1; }
}
```

Mutable Fields

Scala

```
class C:
  val x = 1
  var z = 1
```

Java

```
public class C {
  private final int x = 1;
  private int z = 1;
  public int x() { return x; }
  public int z() { return z; }
  public void z_$eq(int z) { this.z = z; }
}
```

Structured Data

- Tuples: fixed number of heterogeneous items `(1, "hello")`
- Lists: variable number of homogeneous items
  `List(1, 2, 3)` or `1 :: 2 :: 3 :: Nil`
- Immutable and mutable variants
- Pattern matching to decompose structured data into its components

Scala Collections

- Scala collections guide
- `scala.collection`
- `scala.collection.immutable`
- `scala.collection.mutable`
- `java.util` is available
- Scala has arrays `Array[Int]`

Mutability: Fields vs. Data

- Field mutability is different from data mutability
- Java *mutable linked list* by default

```
List<Integer> xs = new List<> ();
final List<Integer> ys = xs; // aliasing
xs.add (4); ys.add (5); // list is mutable through both references
xs = new List<> ();     // reference is mutable
ys = new List<> ();     // fails; reference is immutable
```

- Scala *immutable linked list* by default

```
var xs = List (4, 5, 6)
val ys = xs
xs (1) = 7; ys (1) = 3  // fails; list is immutable
xs = List (0)           // reference is mutable
ys = List ()            // fails; reference is immutable
```

Tuples

💡 Tuples are immutable heterogeneous complex data items

Scala Tuples

```
val p : (Int, String) = (5, "hello")
val x : Int = p(0)
```

Java Pair Class

```
public class Pair<X,Y> {
  final X x;
  final Y y;
  public Pair (X x, Y y) { this.x = x; this.y = y; }
}

Pair<Integer, String> p = new Pair<> (5, "hello");
int x = p.x;
```

Pattern Matching

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

Pattern Matching

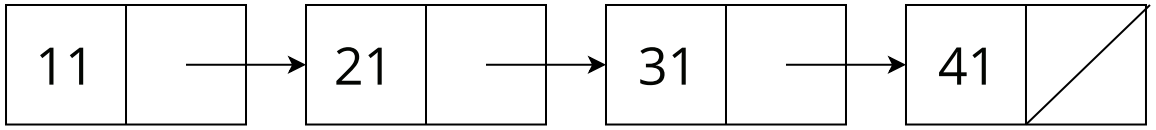
```
def a(p:(Int,Int)) = p match
  case (x,y) => x+y
```

Decomposition with Projections

```
def b(p:(Int,Int)) =
  if p==null then throw MatchError(p)
  val x = p(0)
  val y = p(1)
  x + y
```

Linked Lists

- Scala's `::` is an *infix* `cons` operator for lists



- Scheme

```
(define xs (cons 11 (cons 21 (cons 31 (cons 41 ()))))
```

- Scala

```
val xs = 11 :: (21 :: (31 :: (41 :: Nil))) // List(11, 21, 31, 41)
val xs = 11 :: 21 :: 31 :: 41 :: Nil       // right associative
// method-call style, not encouraged
val xs = Nil.::(41).::(31).::(21).::(11)
```

Operator `cons`: Scheme vs. Scala

Scheme

Scala

- Unlike Scheme `cons`, Scala's `::` requires a list as its right-hand side argument

```
(define x (cons 11 "hello")) // Scheme
```

```
val x = 11 :: "hello" // not Scala, right-hand side of :: must be a list
val x = (11, "hello") // Scala tuples for heterogeneous cons cells
```

- Scala `Nil` is the empty list, shorthand for `List()`

```
(let (emptylist ()))
```

```
val emptylist = Nil // = List()
```

List Constructors

Scheme

```
(list 1 2 (+ 1 2))
```

Scala

```
List (1, 2, 1 + 2)
1 :: 2 :: (1+2) :: Nil
```

List Projections

- Projections extract components of a list: often called `head` and `tail`

Scheme

```
(car xs)
(cdr xs)
```

Scala

```
xs.head
xs.tail
```

Pattern Matching

- Pattern matching branches and binds *pattern variables*

Pattern Matching

```
def f(xs: List[Int]) = xs match
  case Nil   => "List is empty"
  case y::ys => "List is non-empty, head is %d".format (y)
```

Conditionals with Type Tests

```
def g(xs: List[Int]) =
  if xs == Nil then "List is empty"
  else if xs.isInstanceOf[::[Int]] then
    val zs = xs.asInstanceOf[::[Int]]
    val y : Int = zs.head
    val ys : List[Int] = zs.tail
    "List is non-empty, head is %d".format (y)
  else throw MatchError(xs)
```

Pattern Matching

- Nested patterns: patterns can include other patterns

```
def f (xs: List[(Int,String)]) = xs match
  case Nil          => "List is empty"
  case _::Nil       => "List has one element"
  case _::(x,_)::_ => s"The second int is ${x}"

val zs = List ((11,"dog"), (21,"cat"), (31,"pig"))
f(zs)
```

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

Pattern Matching

- Pattern matching vs. Projections

Pattern Matching

```
def f (xs: List[(Int,String)]) = xs match
  case Nil          => "List is empty"
  case _::Nil       => "List has one element"
  case _::(x,_)::_ => s"The second int is ${x}"

val zs = List ((11,"dog"), (21,"cat"), (31,"pig"))
f(zs)
```

Decomposition with Projections

```
def f (xs: List[(Int,String)]) =
  if xs == Nil then "List is empty"
  else if xs.tail == Nil then "List has one element"
  else s"The second int is ${xs.tail.head(0)}"

val zs = List ((11,"dog"), (21,"cat"), (31,"pig"))
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: Recursive Length of List

Imperative

```
def length (xs:List[Int]) : Int =
  var length : int = 0
  var current = xs;
  while current != Nil do
    length = length + 1
    current = current.tail
  length
```

Recursive with Pattern Matching

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

- With parametric polymorphism

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

Recursion

Imperative Iteration

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

- The state of the computation is in mutable variables

Recursive Iteration

```
length (List (1, 2, 3))
--> length (1::(2::(3::Nil)))
--> 1 + length (2::(3::Nil))
--> 1 + (1 + length (3::Nil))
--> 1 + (1 + (1 + length (Nil)))
--> 1 + (1 + (1 + 0))
--> 1 + (1 + 1)
--> 1 + 2
--> 3
```

- The state of the computation is the expression

Appending Lists

Scheme

```
(define (append xs ys)
  (if (equal? xs ())
      ys
      (cons (car xs) (append (cdr xs) ys))))
```

Scala

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

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

- Cons cells created with `1` and `2` in head
- Cons cell `3::Nil` is reused (shared)
- New list, but second part is shared!

Appending Lists

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

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

Recursion

- What does `f` do?

```
def f [X] (xs:List[X]) : List[X] = xs match
  case Nil   => Nil
  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

- Scala combines functional and object-oriented programming
- Builtin support for tuples
- Pattern matching to decompose lists, tuples, and objects into their components
- Favors immutable data and recursion over mutable data and iteration