Scala Programming Language

Mathematical Notes

October 27, 2025

Contents

1 Introduction to Scala

1.1 What is Scala?

Definition 1.1 (Scala Programming Language)**.** *Scala is a general-purpose programming language that combines object-oriented and functional programming paradigms. It runs on the Java Virtual Machine (JVM) and is designed to be concise, elegant, and type-safe.*

Scala was created by Martin Odersky and first released in 2003. Key characteristics include:

- **Object-Oriented** - Everything is an object
- **Functional** - Functions are first-class values
- **Statically Typed** - Type checking at compile time
- **JVM Compatible** - Runs on Java Virtual Machine
- **Concise** - Reduces boilerplate code
- **Type Safe** - Prevents many runtime errors

1.2 Scala Design Principles

1. **Unified Types** - All types inherit from Any
2. **Everything is an Expression** - Statements return values
3. **Immutability by Default** - Encourages functional programming
4. **Pattern Matching** - Powerful control structure
5. **Type Inference** - Compiler infers types when possible
6. **Extensibility** - Easy to add new language constructs

2 Basic Syntax and Types

2.1 Type System

Definition 2.1 (Scala Type Hierarchy)**.** *Scala has a unified type system where all types inherit from Any. The hierarchy includes:*

- ***Any*** *- Root of all types*
- ***AnyVal*** *- Value types (Int, Double, Boolean, etc.)*
- ***AnyRef*** *- Reference types (String, List, custom classes)*
- ***Null*** *- Subtype of all reference types*
- ***Nothing*** *- Subtype of all types*

Example 2.1 (Basic Types).

```scala
// Value types
val intValue: Int = 42
val doubleValue: Double = 3.14159
val booleanValue: Boolean = true
val charValue: Char = 'A'

// Reference types
val stringValue: String = "Hello, Scala!"
val listValue: List[Int] = List(1, 2, 3, 4, 5)

// Type inference
val inferred = "Scala infers this is a String"
val number = 42  // Inferred as Int
```

2.2 Variables and Values

Definition 2.2 (Val vs Var).

- ***val*** - *Immutable reference (like final in Java)*
- ***var*** - *Mutable reference*

Example 2.2 (Variable Declarations).

```scala
// Immutable values
val name = "Alice"
val age = 30
val numbers = List(1, 2, 3)

// Mutable variables
var counter = 0
counter += 1

// Cannot reassign val
// name = "Bob"  // Compilation error

// Can reassign var
var mutableName = "Alice"
mutableName = "Bob"  // OK
```

3 Functions

3.1 Function Definition

Definition 3.1 (Function). *A function in Scala is a first-class value that can be assigned to variables, passed as parameters, and returned from other functions.*

Example 3.1 (Basic Functions).

```scala
// Function definition
def add(x: Int, y: Int): Int = {
    x + y
}

// Single expression function
def multiply(x: Int, y: Int): Int = x * y
```

```scala

// Function with default parameters
def greet(name: String, greeting: String = "Hello"): String = {
    s"$greeting, $name!"
}

// Function with multiple parameter lists
def foldLeft[A, B](list: List[A], initial: B)(f: (B, A) => B): B = {
    list.foldLeft(initial)(f)
}
```

3.2 Higher-Order Functions

Definition 3.2 (Higher-Order Function). *A higher-order function is a function that takes other functions as parameters or returns a function as its result.*

Example 3.2 (Higher-Order Functions).

```scala
// Function as parameter
def applyOperation(x: Int, y: Int, op: (Int, Int) => Int): Int = {
    op(x, y)
}

// Usage
val result1 = applyOperation(5, 3, _ + _)  // 8
val result2 = applyOperation(5, 3, _ * _)  // 15

// Function returning function
def createMultiplier(factor: Int): Int => Int = {
    (x: Int) => x * factor
}

val double = createMultiplier(2)
val triple = createMultiplier(3)

println(double(5))  // 10
println(triple(5))  // 15
```

3.3 Anonymous Functions

Definition 3.3 (Anonymous Function). *An anonymous function (lambda) is a function without a name, defined inline.*

Example 3.3 (Anonymous Functions).

```scala
// Anonymous function syntax
val square = (x: Int) => x * x
val add = (x: Int, y: Int) => x + y

// Using with higher-order functions
val numbers = List(1, 2, 3, 4, 5)
val squares = numbers.map(x => x * x)
val evens = numbers.filter(x => x % 2 == 0)

// Placeholder syntax
```

```
val squares2 = numbers.map(_ * _)
val sum = numbers.reduce(_ + _)
```

4 Collections

4.1 Immutable Collections

Definition 4.1 (Immutable Collection). *An immutable collection cannot be modified after creation. Operations return new collections.*

Example 4.1 (Immutable Collections).

```
// Lists
val list = List(1, 2, 3, 4, 5)
val newList = list :+ 6  // Append
val prepended = 0 +: list  // Prepend

// Vectors (efficient random access)
val vector = Vector(1, 2, 3, 4, 5)
val updated = vector.updated(0, 10)

// Sets
val set = Set(1, 2, 3, 4, 5)
val newSet = set + 6
val removed = set - 1

// Maps
val map = Map("a" -> 1, "b" -> 2, "c" -> 3)
val updatedMap = map + ("d" -> 4)
val removedMap = map - "a"
```

4.2 Collection Operations

Example 4.2 (Common Collection Operations).

```
val numbers = List(1, 2, 3, 4, 5)

// Transformations
val doubled = numbers.map(_ * 2)
val filtered = numbers.filter(_ > 2)
val flattened = List(List(1, 2), List(3, 4)).flatten

// Reductions
val sum = numbers.sum
val product = numbers.product
val max = numbers.max
val min = numbers.min

// Folding
val foldSum = numbers.foldLeft(0)(_ + _)
val foldProduct = numbers.foldRight(1)(_ * _)

// Grouping
val grouped = numbers.groupBy(_ % 2)
```

```
val partitioned = numbers.partition(_ % 2 == 0)

// Chaining operations
val result = numbers
    .filter(_ > 2)
    .map(_ * 2)
    .sum
```

5 Pattern Matching

5.1 Basic Pattern Matching

Definition 5.1 (Pattern Matching). *Pattern matching is a powerful control structure that allows matching values against patterns and extracting data.*

Example 5.1 (Pattern Matching).

```
// Simple pattern matching
def describe(x: Any): String = x match {
    case 1 => "One"
    case 2 => "Two"
    case "hello" => "Greeting"
    case true => "Boolean true"
    case _ => "Something else"
}

// Pattern matching with guards
def categorize(x: Int): String = x match {
    case n if n < 0 => "Negative"
    case n if n == 0 => "Zero"
    case n if n > 0 && n < 10 => "Small positive"
    case _ => "Large positive"
}

// Pattern matching on case classes
case class Person(name: String, age: Int)

def greet(person: Person): String = person match {
    case Person("Alice", age) => s"Hello Alice, you are $age years old"
    case Person(name, age) if age < 18 => s"Hello $name, you are young"
    case Person(name, _) => s"Hello $name"
}
```

5.2 Case Classes

Definition 5.2 (Case Class). *A case class is a special type of class that automatically provides pattern matching, equality, and other useful methods.*

Example 5.2 (Case Classes).

```
// Basic case class
case class Point(x: Int, y: Int)

val point1 = Point(1, 2)
```

```scala
val point2 = Point(1, 2)

// Automatic equality
println(point1 == point2)  // true

// Pattern matching
def describePoint(p: Point): String = p match {
    case Point(0, 0) => "Origin"
    case Point(x, 0) => s"On x-axis at $x"
    case Point(0, y) => s"On y-axis at $y"
    case Point(x, y) => s"Point at ($x, $y)"
}

// Sealed traits for exhaustive matching
sealed trait Shape
case class Circle(radius: Double) extends Shape
case class Rectangle(width: Double, height: Double) extends Shape
case class Triangle(base: Double, height: Double) extends Shape

def area(shape: Shape): Double = shape match {
    case Circle(r) => math.Pi * r * r
    case Rectangle(w, h) => w * h
    case Triangle(b, h) => 0.5 * b * h
}
```

6 Object-Oriented Programming

6.1 Classes and Objects

Definition 6.1 (Class). *A class in Scala is a blueprint for creating objects. It can contain fields, methods, and constructors.*

Example 6.1 (Classes and Objects).

```scala
// Basic class
class Person(val name: String, var age: Int) {
    def greet(): String = s"Hello, I'm $name"

    def haveBirthday(): Unit = {
        age += 1
    }

    override def toString: String = s"Person($name, $age)"
}

// Usage
val person = new Person("Alice", 30)
println(person.greet())
person.haveBirthday()
println(person.age)  // 31

// Companion object
class Counter {
    private var count = 0
```

```scala

    def increment(): Unit = count += 1
    def getCount: Int = count
}

object Counter {
    def apply(): Counter = new Counter()
    def apply(initial: Int): Counter = {
        val c = new Counter()
        c.count = initial
        c
    }
}

// Using companion object
val counter1 = Counter()
val counter2 = Counter(10)
```

6.2 Traits

Definition 6.2 (Trait)**.** *A trait is similar to an interface in Java but can contain concrete methods and fields.*

Example 6.2 (Traits)**.**

```scala
// Basic trait
trait Drawable {
    def draw(): String

    def drawWithBorder(): String = {
        s"[${draw()}]"
    }
}

trait Movable {
    def move(x: Int, y: Int): String
}

// Class implementing traits
class Circle(radius: Double) extends Drawable with Movable {
    def draw(): String = s"Circle with radius $radius"

    def move(x: Int, y: Int): String = s"Moved circle to ($x, $y)"
}

// Usage
val circle = new Circle(5.0)
println(circle.draw())
println(circle.drawWithBorder())
println(circle.move(10, 20))

// Multiple trait inheritance
trait Printable {
    def print(): Unit = println(toString)
}
```

```scala

class Square(side: Double) extends Drawable with Printable {
    def draw(): String = s"Square with side $side"

    override def toString: String = draw()
}

val square = new Square(4.0)
square.print()
```

6.3 Inheritance

Example 6.3 (Inheritance)**.**

```scala
// Base class
abstract class Animal(val name: String) {
    def makeSound(): String

    def introduce(): String = {
        s"I'm $name and I say ${makeSound()}"
    }
}

// Concrete subclass
class Dog(name: String) extends Animal(name) {
    def makeSound(): String = "Woof!"

    def fetch(): String = "Fetching the ball!"
}

// Another subclass
class Cat(name: String) extends Animal(name) {
    def makeSound(): String = "Meow!"

    def purr(): String = "Purring..."
}

// Usage
val dog = new Dog("Buddy")
val cat = new Cat("Whiskers")

println(dog.introduce())
println(cat.introduce())
println(dog.fetch())
println(cat.purr())
```

7 Functional Programming

7.1 Immutability

Definition 7.1 (Immutability)**.** *Immutability means that once a value is created, it cannot be changed. Scala encourages immutable data structures.*

Example 7.1 (Immutable Programming).

```scala
// Immutable data structures
case class Address(street: String, city: String, zipCode: String)
case class Person(name: String, age: Int, address: Address)

val address = Address("123 Main St", "Anytown", "12345")
val person = Person("Alice", 30, address)

// Creating new instances instead of modifying
val olderPerson = person.copy(age = person.age + 1)
val movedPerson = person.copy(
    address = person.address.copy(city = "New City")
)

// Immutable collections
val numbers = List(1, 2, 3, 4, 5)
val doubled = numbers.map(_ * 2)  // Creates new list
val filtered = numbers.filter(_ > 2)  // Creates new list

// Functional transformations
val result = numbers
    .filter(_ % 2 == 0)
    .map(_ * 2)
    .sum
```

7.2 Higher-Order Functions

Example 7.2 (Advanced Higher-Order Functions).

```scala
// Currying
def add(x: Int)(y: Int): Int = x + y
val addFive = add(5)_
println(addFive(3))  // 8

// Partial application
def multiply(x: Int, y: Int, z: Int): Int = x * y * z
val multiplyByTwo = multiply(2, _, _)
val result = multiplyByTwo(3, 4)  // 24

// Function composition
val f = (x: Int) => x * 2
val g = (x: Int) => x + 1
val composed = f compose g
val andThen = f andThen g

println(composed(5))  // f(g(5)) = f(6) = 12
println(andThen(5))   // g(f(5)) = g(10) = 11

// Custom higher-order function
def processList[A, B](list: List[A])(f: A => B): List[B] = {
    list.map(f)
}

val numbers = List(1, 2, 3, 4, 5)
val strings = processList(numbers)(_.toString)
```

```scala
val squares = processList(numbers)(x => x * x)
```

7.3 Monads

Definition 7.2 (Monad)**.** *A monad is a design pattern that allows chaining operations while handling side effects. Common monads in Scala include Option, Try, and Future.*

Example 7.3 (Option Monad)**.**

```scala
// Option for handling null values
def divide(a: Int, b: Int): Option[Int] = {
    if (b != 0) Some(a / b) else None
}

// Using Option
val result1 = divide(10, 2)  // Some(5)
val result2 = divide(10, 0)  // None

// Pattern matching with Option
def describeResult(result: Option[Int]): String = result match {
    case Some(value) => s"The result is $value"
    case None => "Division by zero!"
}

// Monadic operations
val numbers = List(1, 2, 3, 4, 5)
val results = numbers.map(n => divide(10, n))
val validResults = results.collect { case Some(x) => x }

// FlatMap for chaining
def safeDivide(a: Int, b: Int): Option[Int] = divide(a, b)
def safeAdd(a: Int, b: Int): Option[Int] = Some(a + b)

val chained = safeDivide(10, 2).flatMap(x => safeAdd(x, 5))
println(chained)  // Some(10)

// For-comprehension (syntactic sugar for flatMap)
val result = for {
    x <- safeDivide(10, 2)
    y <- safeAdd(x, 5)
} yield y
println(result)  // Some(10)
```

8 Error Handling

8.1 Try Monad

Definition 8.1 (Try)**.** *Try is a monad for handling exceptions in a functional way. It can be either Success(value) or Failure(exception).*

Example 8.1 (Error Handling with Try)**.**

```scala
import scala.util.{Try, Success, Failure}

```

```scala
// Function that might throw an exception
def riskyOperation(x: Int): Int = {
    if (x < 0) throw new IllegalArgumentException("Negative number")
    x * 2
}

// Using Try
val result1 = Try(riskyOperation(5))   // Success(10)
val result2 = Try(riskyOperation(-1))  // Failure(IllegalArgumentException)

// Pattern matching with Try
def handleResult(result: Try[Int]): String = result match {
    case Success(value) => s"Success: $value"
    case Failure(exception) => s"Error: ${exception.getMessage}"
}

// Monadic operations
val processed = Try(riskyOperation(5))
    .map(_ + 10)
    .map(_ * 2)

// Recovering from errors
val recovered = Try(riskyOperation(-1))
    .recover {
        case _: IllegalArgumentException => 0
    }

// For-comprehension with Try
val result = for {
    x <- Try(riskyOperation(5))
    y <- Try(riskyOperation(3))
} yield x + y
```

8.2 Either Monad

Example 8.2 (Either for Error Handling)**.**

```scala
// Either for explicit error handling
def divideEither(a: Int, b: Int): Either[String, Int] = {
    if (b == 0) Left("Division by zero")
    else Right(a / b)
}

// Using Either
val result1 = divideEither(10, 2)  // Right(5)
val result2 = divideEither(10, 0)  // Left("Division by zero")

// Pattern matching
def describeEither(result: Either[String, Int]): String = result match {
    case Right(value) => s"Result: $value"
    case Left(error) => s"Error: $error"
}

```

```scala
// Monadic operations
val processed = divideEither(10, 2)
    .map(_ * 2)
    .map(_ + 10)

// FlatMap for chaining
def safeDivide(a: Int, b: Int): Either[String, Int] = divideEither(a, b)
def safeAdd(a: Int, b: Int): Either[String, Int] = Right(a + b)

val chained = safeDivide(10, 2).flatMap(x => safeAdd(x, 5))
```

9 Concurrency

9.1 Futures

Definition 9.1 (Future)**.** *A Future represents a value that will be available at some point in the future, typically as a result of an asynchronous computation.*

Example 9.1 (Futures)**.**

```scala
import scala.concurrent.{Future, ExecutionContext}
import scala.concurrent.ExecutionContext.Implicits.global

// Basic Future
def slowComputation(): Future[Int] = Future {
    Thread.sleep(1000)
    42
}

// Using Future
val future = slowComputation()
future.onComplete {
    case scala.util.Success(value) => println(s"Result: $value")
    case scala.util.Failure(exception) => println(s"Error: $exception")
}

// Transforming Futures
val transformed = future.map(_ * 2)
val flatMapped = future.flatMap(x => Future(x + 10))

// Combining Futures
val future1 = Future(1)
val future2 = Future(2)
val combined = for {
    x <- future1
    y <- future2
} yield x + y

// Awaiting results (blocking)
import scala.concurrent.Await
import scala.concurrent.duration._

val result = Await.result(combined, 5.seconds)
println(result)  // 3
```

9.2 Akka Actors

Example 9.2 (Actor Model).

```scala
import akka.actor.{Actor, ActorSystem, Props}

// Simple Actor
class Greeter extends Actor {
    def receive = {
        case "hello" => println("Hello there!")
        case "goodbye" => println("Goodbye!")
        case _ => println("Unknown message")
    }
}

// Actor with state
class Counter extends Actor {
    var count = 0

    def receive = {
        case "increment" => count += 1
        case "decrement" => count -= 1
        case "get" => sender() ! count
    }
}

// Usage
val system = ActorSystem("MySystem")
val greeter = system.actorOf(Props[Greeter], "greeter")
val counter = system.actorOf(Props[Counter], "counter")

greeter ! "hello"
counter ! "increment"
counter ! "increment"
counter ! "get"
```

10 Advanced Features

10.1 Implicit Conversions

Definition 10.1 (Implicit Conversion). *An implicit conversion automatically converts one type to another when needed.*

Example 10.1 (Implicit Conversions).

```scala
// Implicit conversion
implicit def intToString(x: Int): String = x.toString

// Usage
val str: String = 42  // Automatically converted

// Implicit parameters
def greet(name: String)(implicit greeting: String): String = {
    s"$greeting, $name!"
}

```

```scala
implicit val defaultGreeting = "Hello"

val message = greet("Alice")  // Uses implicit greeting

// Implicit classes (extension methods)
implicit class RichInt(x: Int) {
    def isEven: Boolean = x % 2 == 0
    def isOdd: Boolean = !isEven
}

val number = 42
println(number.isEven)  // true
println(number.isOdd)   // false
```

10.2 Type Classes

Example 10.2 (Type Classes).

```scala
// Type class definition
trait Show[A] {
    def show(a: A): String
}

// Type class instances
implicit val intShow: Show[Int] = new Show[Int] {
    def show(a: Int): String = s"Int: $a"
}

implicit val stringShow: Show[String] = new Show[String] {
    def show(a: String): String = s"String: $a"
}

// Type class usage
def printShow[A](a: A)(implicit show: Show[A]): Unit = {
    println(show.show(a))
}

// Usage
printShow(42)       // Int: 42
printShow("hello") // String: hello

// Context bounds syntax
def printShow2[A: Show](a: A): Unit = {
    println(implicitly[Show[A]].show(a))
}
```

10.3 Macros

Example 10.3 (Macros (Conceptual)).

```scala
// Macro example (simplified)
import scala.reflect.macros.blackbox.Context
import scala.language.experimental.macros

def assert(condition: Boolean): Unit = macro assertImpl
```

```scala

def assertImpl(c: Context)(condition: c.Expr[Boolean]): c.Expr[Unit] = {
    import c.universe._
    val q"$expr" = condition
    c.Expr[Unit](q"""
        if (!$condition) {
            throw new AssertionError("Assertion failed: " + ${expr.
                toString})
        }
    """)
}

// Usage
val x = 5
assert(x > 0)  // Compile-time assertion
```

11 Scala Collections Deep Dive

11.1 Performance Characteristics

Definition 11.1 (Collection Performance). *Different Scala collections have different performance characteristics for various operations.*

Example 11.1 (Collection Performance).

```scala
// List - O(1) prepend, O(n) append
val list = List(1, 2, 3)
val prepended = 0 +: list  // O(1)
val appended = list :+ 4   // O(n)

// Vector - O(log n) for most operations
val vector = Vector(1, 2, 3)
val updated = vector.updated(0, 10)  // O(log n)

// Array - O(1) random access
val array = Array(1, 2, 3, 4, 5)
val element = array(2)  // O(1)

// Set - O(log n) for most operations
val set = Set(1, 2, 3, 4, 5)
val contains = set.contains(3)  // O(log n)

// Map - O(log n) for most operations
val map = Map("a" -> 1, "b" -> 2, "c" -> 3)
val value = map("b")  // O(log n)
```

11.2 Streams and Lazy Evaluation

Example 11.2 (Lazy Collections).

```scala
// Stream - lazy collection
val stream = Stream.from(1)
val firstTen = stream.take(10).toList

```

```scala
// Lazy evaluation
def expensiveComputation(x: Int): Int = {
    println(s"Computing for $x")
    x * x
}

val numbers = List(1, 2, 3, 4, 5)
val lazyResults = numbers.view.map(expensiveComputation)
// Nothing computed yet

val firstResult = lazyResults.head  // Only computes for 1
val allResults = lazyResults.toList  // Computes for all

// Lazy initialization
lazy val expensiveValue = {
    println("Computing expensive value")
    42
}

// Only computed when first accessed
println(expensiveValue)
```

12 Testing

12.1 ScalaTest

Example 12.1 (Testing with ScalaTest).

```scala
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should.Matchers

class CalculatorSpec extends AnyFlatSpec with Matchers {

    "A Calculator" should "add two numbers correctly" in {
        val calculator = new Calculator()
        calculator.add(2, 3) should be (5)
    }

    it should "multiply two numbers correctly" in {
        val calculator = new Calculator()
        calculator.multiply(4, 5) should be (20)
    }

    it should "handle division by zero" in {
        val calculator = new Calculator()
        assertThrows[ArithmeticException] {
            calculator.divide(10, 0)
        }
    }
}

class Calculator {
    def add(a: Int, b: Int): Int = a + b
    def multiply(a: Int, b: Int): Int = a * b
```

```scala
    def divide(a: Int, b: Int): Int = {
        if (b == 0) throw new ArithmeticException("Division by zero")
        a / b
    }
}
```

13 Build Tools

13.1 SBT (Scala Build Tool)

Example 13.1 (SBT Configuration).

```scala
// build.sbt
name := "scala-project"
version := "1.0"
scalaVersion := "2.13.8"

libraryDependencies ++= Seq(
    "org.scalatest" %% "scalatest" % "3.2.10" % Test,
    "com.typesafe.akka" %% "akka-actor" % "2.6.18"
)

// Common SBT commands:
// sbt compile
// sbt test
// sbt run
// sbt package
// sbt clean
```

14 Best Practices

14.1 Code Style

1. Use `val` instead of `var` when possible
2. Prefer immutable collections
3. Use pattern matching instead of if-else chains
4. Leverage type inference
5. Use meaningful names for variables and functions
6. Write pure functions when possible
7. Use case classes for data structures
8. Prefer composition over inheritance

14.2 Functional Programming Guidelines

1. Avoid side effects
2. Use higher-order functions
3. Prefer immutable data structures
4. Use monads for error handling
5. Leverage pattern matching
6. Write small, focused functions
7. Use type classes for polymorphism

15 Conclusion

Scala is a powerful programming language that successfully combines object-oriented and functional programming paradigms. Its key strengths include:

- **Type Safety** - Compile-time error detection
- **Conciseness** - Reduced boilerplate code
- **Functional Programming** - First-class functions and immutability
- **Object-Oriented** - Classes, traits, and inheritance
- **JVM Compatibility** - Access to Java ecosystem
- **Pattern Matching** - Powerful control structure
- **Type Inference** - Less verbose type annotations
- **Extensibility** - Easy to add new language constructs

Scala is particularly well-suited for:

- Large-scale applications
- Data processing and analytics
- Concurrent and distributed systems
- Domain-specific languages
- Functional programming enthusiasts

The language continues to evolve with new features and improvements, making it an excellent choice for modern software development.