Software Construction

Notes

Isaac Metthez

December 18, 2025

Notes summarizing core topics from EPFL courses

CS-214 – Software Construction

Latest version:
Software Construction Notes

Contents

1 Functional programming

1.1 Definition

Main programming paradigms:

- Imperative, Functional, Logic programming

Orthogonal to it:

- Object-oriented programming

Scaling up "Von Neumann" bottleneck: one tends to conceptualize data structures word-by-word.

Immutable variables Variables that cannot be modified after their initial assignment. Immutability simplifies reasoning about programs and enables safe concurrent execution.

1.2 Scala

This section covers Scala 3 syntax and features.

1.2.1 Build tool

The standard build tool for Scala is **SBT**. Used to compile, test, run, and package Scala projects.

```
# Init new project
sbt new scala/scala3.g8
# Compile project
sbt compile
# Run project
sbt run
# Test project
sbt test
sbt ~test # auto recompile and test on file changes
# Run REPL
sbt console
```

1.2.2 Comment

```
// Single line comment
def f(x: Int): Int = x * x // comment at end of line

/* Scoped comment  */
def h(x: Int): Int =
  /* Comment with
  multilines */
  x * x

def g(x: Int): Int =
  x + x /* comment in expression */ + 1


/**
  * Doc comment
  *
  */
def doc(x: Int): Int = x * x
```

1.2.3 Variables

```
// Declaration in a scope (Type can be omitted and is inferred by the compiler)
var x : <Type> = <expression> // mutable variable evaluated once (call by value)
val y : <Type> = <expression> // immutable variable evaluated once (call by value)
def z : <Type> = <expression> // function without argument, re-evaluated at each use (call by
↪  name)
lazy val w : <Type> = <expression> // immutable variable evaluated once at first use (call by
↪  need)
```

1.2.4 Scope

- Indentation
- {}
- `end` *(just for markup)*

In these notes indentation is preferred. But both can be mixed and are completely equivalent.

```
// { } Scope
def f(x: Int): Int = {
  val y = x * x
  y + 1
}
end f // just for markup

case class Cls(v: Int){
  def sum: Int = {
    val w = v + 10
    w * 2
  }
}
// Indentation scope
def g(x: Int): Int =
  val y = x * x
  y + 1

case class Cls(v: Int):
  def sum: Int =
    val w = v + 10
    w * 2
// end (just for markup)
end Cls // just for markup
```

1.2.5 Conditional expressions

In Scala, `if-else` is an expression (returns a value), unlike Java/C# where it's a statement.

```
// Can be assigned to a variable
val max = if a > b then a else b

// Basic syntax
if <condition> then
  <expression>
else
  <expression>
```

1.2.6 Pattern matching

```
<expression> match
  case <pattern 1> => <expression>
  case <pattern 2> => <expression>
```

Pattern Types **1. Literal patterns:**

```
x match
  case 0 => "zero"
  case 1 => "one"
  case "hello" => "greeting"
  case true => "yes"
  case () => "unit"
  case null => "null value"
```

2. Variable patterns:

```
x match
  case n => s"value is $n"  // binds x to variable n
  case _ => "wildcard"       // wildcard, ignores value
```

3. Constructor patterns (case classes):

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

point 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)"
```

4. Tuple patterns:

```
pair match
  case (0, 0) => "origin"
  case (x, 0) => s"x = $x"
  case (0, y) => s"y = $y"
  case (x, y) => s"($x, $y)"

// Nested tuples
triple match
  case (x, (y, z)) => s"x=$x, y=$y, z=$z"
```

5. Type patterns:

```
value match
  case s: String => s"string: $s"
  case i: Int => s"integer: $i"
  case d: Double => s"double: $d"
  case _: List[_] => "some list"
  case _ => "unknown type"
```

6. Pattern with @-binding (variable binding):

```
// Bind the entire pattern to a variable with @
point match
  case p @ Point(0, 0) => s"origin point: $p"
  case p @ Point(x, y) if x == y => s"diagonal point: $p at ($x,$y)"

list match
  case xs @ List(1, 2, _*) => s"list starting with 1,2: $xs"

// Useful for nested patterns
person match
  case Person(name, addr @ Address(city, _)) =>
    s"$name lives in $city, full address: $addr"
```

7. Sequence patterns:

```
list match
  case List() => "empty"
  case List(x) => s"one element: $x"
  case List(x, y) => s"two elements: $x, $y"
  case List(1, 2, 3) => "exact list [1,2,3]"
  case x :: xs => s"head: $x, tail: $xs"
```

```
  case List(1, 2, _*) => "starts with 1, 2"
  case List(_, _, 3) => "third element is 3"
```

8. Guards (conditional patterns):

```
x match
  case n if n < 0 => "negative"
  case n if n > 0 => "positive"
  case _ => "zero"

point match
  case Point(x, y) if x == y => "on diagonal"
  case Point(x, y) if x > y => "above diagonal"
  case Point(x, y) if x < y => "below diagonal"
```

9. Alternative patterns (OR):

```
x match
  case 0 | 1 => "zero or one"
  case 2 | 3 | 5 | 7 => "small prime"
  case _ => "other"
```

10. Typed patterns with generics:

```
value match
  case list: List[Int] => "list of integers"  //  type erasure!
  case map: Map[String, Int] => "string to int map"  //  type erasure!
  case opt: Option[_] => "some option"
```

Warning: Due to type erasure, generic type parameters are not checked at runtime:

```
val x: Any = List("a", "b")

x match
  case list: List[Int] => println("integers")  //  Matches! (wrong)
  case _ => println("other")
// Prints "integers" even though it's List[String]!
```

11. Constant patterns:

```
val MaxValue = 100

x match
  case MaxValue => "at maximum"  // uses the constant
  case other => s"value: $other"
```

12. Infix operation patterns:

```
list match
  case first :: second :: rest => s"at least 2 elements: $first, $second"
  case head :: tail => s"head: $head"
  case Nil => "empty"
```

13. Pattern matching in variable declarations:

```
// Destructuring tuples
val (x, y) = (1, 2)

// Destructuring case classes
val Point(px, py) = Point(3, 4)

// Destructuring lists
val head :: tail = List(1, 2, 3)  // can fail if list is empty

// Safer with pattern matching
val list = List(1, 2, 3)
val result = list match
  case h :: t => Some(h, t)
```

```
  case Nil => None
```

Pattern Matching Best Practices

- Use `case _` as a catch-all to avoid `MatchError`
- Prefer sealed traits for exhaustiveness checking
- Use guards sparingly - complex logic should be in separate functions
- `@`-binding is useful when you need both the whole and parts
- Order matters: patterns are checked top-to-bottom, first match wins

Exhaustiveness checking with sealed traits:

```
sealed trait Shape
case class Circle(radius: Double) extends Shape
case class Rectangle(width: 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
// Compiler warns if we forget a case!
```

1.2.7 Functions

Base

```
// Define
def fun(v: <Type>): <Type> =
  <expression>

val fun = (v: <Type>) => <expression>

// Call
fun(<expression>)

// Simpler call with block syntax for single parameter functions
// Warning: The expression must be on a separate line after the ":"
fun:
  <expression>

// Methods can be call with infix notation
class Cls:
  infix def process(x: Int): Int = x * 2 // Infix method

val c = Cls()

// Normal call
c.process(5)
// Infix call
c process 5
// Because there is only one parameter
c.process:
  5

// Function type
<Type> => <expression return type>
```

Closures A closure is a function that captures variables from its surrounding lexical scope.

```
def makeAdder(x: Int): Int => Int =
  (y: Int) => x + y  // captures 'x' from outer scope
```

```
val add5 = makeAdder(5)
add5(3)  // returns 8
```

The function "(y: Int) => x + y" is a closure because it "closes over" the variable "x" from the enclosing scope.

Var args

```
// Define
def fun(v: <Type>*): <Type> =
  <expression>

// Call
// Normal call
fun(<expression 1>, <expression 2>, ...)

// Explode a sequence
fun(seq*)
```

Currying

```
// Define
def fun(v: <Type>)(w: <Type>): <Type> =
  <expression>

def fun(v: <Type>)(w: <Type>): <Type> =
  def innerFun(v: <Type>)(w: <Type>): <Type> =
    <expression>
  <expression> // last expression is the return

val fun = (v: <Type>) => (w: <Type>) => <expression>

// Call
fun(<expression 1>)(<expression 2>)

// Function type
<Type> => <Type> => <expression return type>
```

Call by value or by name By default, function parameters in Scala are passed by value. But sometimes we want explicitly to pass by name. For instance the for the Try monad.

```
// Example Try monad
object Try:
  def apply[A](expr: => A): Try[A] = // expr is call by name
    try Success(expr)
    catch case e: Exception => Failure(e)

// then
val result = Try:
  // some computation that may throw
  computeSomething()

// or else
val result2 = Try(computeSomething()) // is also valid

// This is different from just passing a function by value
// Example passing by value function
object TryByValue:
  def apply[A](expr: () => A): Try[A] = // expr is call by value
    try Success(expr())
    catch case e: Exception => Failure(e)

// then
val result = TryByValue(() =>
```

```
  // some computation that may throw
  computeSomething())
```

Functions as objects The function type `A => B` is an abbreviation for `scala.Function1[A, B]`.

```
package scala
trait Function1[A, B]:
  def apply(x: A): B

val f = (x: Int) => x * x
f(7)

// => as object
val f = new Function1[Int, Int]:
  def apply(x: Int) = x * x
f.apply(7)
```

Note that a method such as:

```
def f(x: Int): Boolean = ...
```

is not itself a function value. But if `f` is used in a place where a `Function` type is expected, it is converted automatically to a function value.

Partial Functions A **partial function** is a function that is not defined for all inputs of its input type.

```
// Regular function - defined for ALL Int
val double: Int => Int = x => x * 2

// Partial function - only defined for positive Int
val sqrt: PartialFunction[Int, Double] =
  case x if x >= 0 => Math.sqrt(x)
  // Throws MatchError for negative numbers

sqrt.isDefinedAt(4)   // true
sqrt.isDefinedAt(-1)  // false
```

Main use case: with `collect` to filter and map simultaneously. See section 1.2.35 for more collection operations.

```
val mixed: List[Any] = List(1, "hello", 2, "world")

// Extract only strings and uppercase them
val strings = mixed.collect:
  case s: String => s.toUpperCase
// Result: List("HELLO", "WORLD")

// Equivalent to:
mixed.filter(_.isInstanceOf[String])
     .map(_.asInstanceOf[String].toUpperCase)
```

The `case ...` syntax (with or without braces) creates a partial function that automatically filters out unmatched cases.

1.2.8 Types

Type alias can be defined with `type` keyword.

```
type Word = String // type alias
type Pair[A, B] = (A, B) // generic type alias
```

```
// Opaque type alias, hides implementation details (means that outside code cannot rely on the
↪  underlying type)
opaque type Meter = Double
```

1.2.9 Classes

```
class Cls(v: <Type>, w: <Type>):
  private val max = if v < w then w else v
  private def mod = v % w
  def sum = this.v + this.w

  // auxiliary constructor
  def this(a: <Type>) =
    this(a, a) // call primary constructor

  // require (precondition)
  require(v > 0, "v must be positive") // if false: IllegalArgumentException

  // assert (internal check)
  def sqrt =
    val r = math.sqrt(w)
    assert(r >= 0, "r must be non-negative") // if false: AssertionError
```

Default inheritance All classes in Scala implicitly extend the `AnyRef` class (equivalent to `java.lang.Object` in Java) if no other superclass is specified.

1.2.10 Special methods on Classes

Apply The `apply` method allows instances of a class to be called like functions.

```
class Adder(factor: Int):
  def apply(x: Int): Int = x + factor
val add5 = Adder(5)
val result = add5(10) // Calls add5.apply(10), result is 15
```

Unapply The `unapply` method is used in pattern matching to extract values from an object.

```
object Even:
  def unapply(x: Int): Option[Int] = // Must return Option type
    if x % 2 == 0 then Some(x / 2) else None

val number = 8

number match
  case Even(half) => println(s"$number is even, half is $half")
  case _          => println(s"$number is odd")
```

UnapplySeq The `unapplySeq` method is used for pattern matching on sequences, allowing extraction of variable-length sequences.

```
object Words:
  def unapplySeq(s: String): Option[Seq[String]] =
    val words = s.split(" ").toSeq
    if words.nonEmpty then Some(words) else None
val sentence = "Hello Scala world"
sentence match
  case Words(first, second, rest @ _*) =>
    println(s"First word: $first, Second word: $second, Rest: $rest")
  case _ =>
    println("No words found")
```

Update The `update` method allows instances of a class to be updated using array-like syntax.

```
class MutableArray(size: Int):
  private val data = new Array[Int](size)
  def apply(index: Int): Int = data(index)
  def update(index: Int, value: Int): Unit = data(index) = value

val arr = MutableArray(5)
arr(0) = 10 // Calls arr.update(0, 10)
val value = arr(0) // Calls arr.apply(0), value is 10
```

Unary operators The unary operators in Scala are special methods that allow you to define custom behavior for unary operations on your classes. The supported unary operators are `+`, `-`, `!`, and `~`.

```
class Counter(var value: Int):
  def unary_+ : Counter = Counter(value + 1)
  def unary_- : Counter = Counter(value - 1)
  def unary_! : Boolean = value == 0
  def unary_~ : Counter = Counter(-value)
val counter = Counter(5)
val incremented = +counter // Calls counter.unary_+
val decremented = -counter // Calls counter.unary_-
val isZero = !counter    // Calls counter.unary_!
val negated = ~counter   // Calls counter.unary_~

val counter += 1 // Calls counter.update(counter.value + 1) // Not unary operator
val counter -= 1 // Calls counter.update(counter.value - 1) // Not unary operator
```

Hierarchy/Inheritance

```
abstract class AbsCls: // superclass, base class of SpecCls
  def move: Int
  def bass: Int = 0

class SpecCls extends AbsCls: // subclass
  def move = 10
  override def bass = 10

object SingletonCls extends AbsCls:
  def move = 20

// Companion (object and class with the same name)
class IntSet(val head: Int, val tail: Option[IntSet])
object IntSet:
  def singleton(x: Int) = NonEmpty(x, Empty, Empty)
  // factory method
  def apply(x : Int, tail: Option[IntSet]): IntSet =
    new IntSet(x, tail)  // call constructor of IntSet (Same principle for case classes)


// Programs
// similar to Java
object Hello:
  def main(args: Array[String]): Unit = println("hello world!")
// > scala Hello

// syntactic sugar
@main def nameProgram(name: String, n: Int) =
  println(s"Name: $name, $n")
// > scala nameProgram test 123
```

The base hierarchy:

- **Any** (base of all types): methods `==`, `!=`, `equals`, `hashCode`, `toString`
- **AnyRef** (alias of `java.lang.Object`): reference types
- **AnyVal**: base type of all primitive types

The bottom:

- **Nothing**: to signal abnormal termination; as element type of empty collections

1.2.11 Traits

```
// Traits
trait Planar:
  def h: Int
  def w: Int
  def surface = h * w

class SpecTCls extends AbsCls, Planar:
  def move = 42

// with is possible (older syntax version)
class SpecTCls extends AbsCls with Planar:
  def move = 42
```

Extension Methods Extension methods add new methods to existing types without modifying their source code. In Scala 3, use the `extension` keyword.

```
// Basic syntax
extension (s: String)
  def shout: String = s.toUpperCase + "!"
  def repeat(n: Int): String = s * n

"hello".shout      // "HELLO!"
"ab".repeat(3)     // "ababab"
```

Generic Extensions Extensions can be parameterized with type parameters.

```
extension [T](xs: List[T])
  def second: Option[T] = xs match
    case _ :: x :: _ => Some(x)
    case _ => None

  def intersperse(sep: T): List[T] = xs match
    case Nil => Nil
    case head :: Nil => List(head)
    case head :: tail => head :: sep :: tail.intersperse(sep)

List(1, 2, 3).second           // Some(2)
List(1, 2, 3).intersperse(0)   // List(1, 0, 2, 0, 3)
```

Extensions with Type Bounds

```
extension [T: Ordering](xs: List[T])
  def sortedDesc: List[T] = xs.sorted.reverse
  def minOption: Option[T] = if xs.isEmpty then None else Some(xs.min)

List(3, 1, 2).sortedDesc  // List(3, 2, 1)
```

Collective Extensions Group multiple extensions in a single block.

```
extension (x: Int)
  def isEven: Boolean = x % 2 == 0
  def isOdd: Boolean = !x.isEven
  def square: Int = x * x
```

```
  infix def divides(y: Int): Boolean = y % x == 0

4.isEven       // true
3.square       // 9
2 divides 10   // true (infix notation)
```

1.2.12 Case Classes

Case classes are special classes in Scala that are optimized for immutable data structures and pattern matching. As record in java or in C# they provide a concise syntax for defining classes that primarily hold data. And they come with several useful methods automatically generated by the compiler. The methods generated are: `equals`, `hashCode`, `toString`, and `copy`.

```
// Case class
case class Point(x: Int, y: Int):
  def move(dx: Int, dy: Int) = Point(x + dx, y + dy)

// Usage
val p1 = Point(2, 3)
val p2 = p1.move(1, -1)

// Pattern matching
p2 match
  case Point(3, 2) => println("Moved correctly")
  case _           => println("Something went wrong")$

// Copy with modification
val p3 = p2.copy(y = 5) // x remains the same

// Enums with datas are case classes
enum Shape:
  case Circle(radius: Double)
  case Rectangle(width: Double, height: Double)

val recCopy = Shape.Rectangle(4.0, 5.0).copy(width = 10.0)
```

1.2.13 Operator overloading

```
extension (c: Cls)
  def +(c2: Cls): Cls = c.add(c2)
  def *(c2: Cls): Cls = c.mul(c2)

  infix def min(that: Rational) = ???

a + b   // instead of a.+(b)
a * b
a min b
```

Operator precedence Determined by the first character:

```
(all letters)
|
^
&
< >
= !
:
+ -
* / %
(all other special characters)
```

1.2.14 Exceptions

```
// class SpecificException(msg: String) extends Exception(msg)

// Raise
throw Exc // the type of the expression is Nothing

// Catch
try
  <expression>
catch
  case e: IOException => println("I/O error")
  case e: SpecificException => println("Specific exception")
  case e: Exception   => println(s"Error: ${e.getMessage}")
  case _              => println("Unknown")
finally
  println("Always executed")
```

1.2.15 Packages

At the top of a file:

```
package test.sample

object Test1
object Test2

// Test1 is in test.sample
```

1.2.16 Imports

```
import test.sample.Test1
import test.sample.{Test1, Test2}
import test.sample.*
// possible to import from a package or an object
```

1.2.17 Package

```
package samppe.example
// same as
package sample
package example
```

1.2.18 Enums

```
// Base
enum ColorEncoding:
  case RGB(r: Int, g: Int, b: Int)
  case HSL(h: Double, s: Double, l: Double)
  case YUV(y: Int, uv: Int)

// unbox
import ColorEncoding.*

// enum is a class
enum Direction(val dx: Int, val dy: Int):
  case Right extends Direction( 1, 0)
  case Up    extends Direction( 0, 1)
  case Left  extends Direction(-1, 0)
  case Down  extends Direction( 0,-1)
  def leftTurn = Direction.values((ordinal + 1) % 4)

// equivalent sketch
abstract class Direction(val dx: Int, val dy: Int):
```

```
  def leftTurn = Direction.values((ordinal + 1) % 4)
object Direction:
  val Right = new Direction( 1, 0)
  val Up    = new Direction( 0, 1)
  val Left  = new Direction(-1, 0)
  val Down  = new Direction( 0,-1)
```

1.2.19 Access modifiers

Access modifiers define the visibility and inheritance rules for classes, traits, methods, and variables in Scala. They control which parts of the code can access or extend a given element.

Main categories

- **Access control**: restricts who can see or use an element.
- **Inheritance control**: restricts how an element can be extended or overridden.

Access control modifiers

- `private`: visible only within the class or object that defines it.
- `protected`: visible in subclasses.
- `public` (default): visible everywhere.
- `private[package]`: visible within a given package and its subpackages.

Inheritance control modifiers

- `sealed`: all subclasses must be defined in the same source file. Ensures that pattern matching can be checked for exhaustiveness.
- `final`: prevents inheritance or overriding.
- `abstract`: marks a class that cannot be instantiated directly and may define abstract members.

Example

```
sealed abstract class Shape

private final class Circle(r: Double) extends Shape

protected case class Rectangle(w: Double, h: Double) extends Shape

// Access limited to package "geometry"
private[geometry] class Helper
```

1.2.20 Inline

The `inline` modifier in Scala is used to suggest to the compiler that a method or value should be inlined at the call site. This can improve performance by eliminating the overhead of a method call, especially for small methods that are called frequently. Transparent inline methods allow the compiler to infer more precise types based on the arguments provided at the call site.

```
// Inline
inline def fun(x: Int): Int = x * x
inline val constant = 42

// Usage
val result = fun(5) + constant

// Transparent inline
transparent inline def max[A](x: A, y: A)(using ord: Ordering[A]): A =
```

```
  if ord.gt(x, y) then x else y

  // Usage
val m = max(10, 20) // inferred as Int
```

1.2.21 Infix

Infix notation allows methods with a single parameter to be called without the dot and parentheses, making the code more readable and resembling natural language. This has already been shown in section 1.2.13.

```
class Rational(n: Int, d: Int):
  def +(that: Rational): Rational = ???
  infix def -(that: Rational): Rational = ???
val r1 = Rational(1, 2)
val r2 = Rational(3, 4)
val sum = r1 + r2       // calls r1.+(r2)
val difference = r1 - r2 // calls r1.-(r2)
```

1.2.22 Specifications

Scala provides built-in mechanisms for expressing formal specifications within the code itself, following the Design by Contract paradigm.

Preconditions with `require`: The `require` statement defines conditions that must be true before a function executes. It validates input parameters and throws an `IllegalArgumentException` if the condition fails.

```
def sqrt(x: Double): Double =
  require(x >= 0, "Cannot compute square root of negative number")
  math.sqrt(x)
```

Postconditions with `ensuring`: The `ensuring` clause specifies conditions that must hold after function execution, validating the returned result.

```
def divide(numerator: Int, denominator: Int): Double =
  require(denominator != 0, "Denominator cannot be zero")
  val result = numerator.toDouble / denominator
  result
.ensuring(result => !result.isNaN && !result.isInfinite)
```

Assertions with `assert`: The `assert` statement checks invariants during execution, useful for debugging and verifying internal consistency.

```
def factorial(n: Int): Int =
  require(n >= 0, "Factorial requires non-negative input")

  if n == 0 then 1
  else
    val result = n * factorial(n - 1)
    assert(result > 0, "Factorial overflow detected")
    result
```

Write signatures without implementation: Abstract methods in traits or abstract classes can define specifications without providing implementations. But sometimes it is useful to provide a default implementation that throws an exception to indicate that the method should be overridden.

```
def withoutImplementation(x: Int): Int =
  ??? // throws NotImplementedError at runtime
```

Sometimes we do not want to throw an exception but indicate that the method is not implemented. For instance to continue test execution. A possible implementation is:

```
def TODO[A]: A = null.asInstanceOf[A]
```

1.2.23 Unit testing

Scala has several popular testing frameworks, including ScalaTest, Specs2, and MUnit. These frameworks provide tools for writing and running unit tests, as well as features for assertions, mocking, and test organization. MUnit is used here as example.

```
import munit.FunSuite
class MyTests extends FunSuite:
  test("addition works"):
    assertEquals(2 + 2, 4)

  test("string concatenation"):
    val result = "Hello, " + "world!"
    assertEquals(result, "Hello, world!")
```

1.2.24 Property-Based Testing with ScalaCheck

ScalaCheck automatically generates test cases to verify properties across many inputs, rather than testing specific examples.

Basic Properties Use `forAll` to test properties with automatically generated inputs:

```
import org.scalacheck.Prop.forAll

forAll:
 (a: Int, b: Int) =>
  a + b == b + a  // commutativity

forAll:
 (x: Int) =>
  x + 1 - 1 == x  // inverse operations
```

Preconditions Use `==>` to express conditional properties:

```
forAll:
  (l: List[Int]) =>
  (l.nonEmpty) ==> (l.head :: l.tail == l)

forAll:
  (x: Int) =>
  (x != Int.MaxValue) ==> (x + 1 > x)
```

Custom Generators Create generators for specific data types:

```
import org.scalacheck.Gen

// Generate values from a range
val diceRoll: Gen[Int] = Gen.choose(1, 6)

// Generate from specific values
val vowel: Gen[Char] = Gen.oneOf('a', 'e', 'i', 'o', 'u')

// Generate lists of specific size
val shortList: Gen[List[Int]] = Gen.listOfN(3, Gen.posNum[Int])
```

```scala
// Custom data structures
case class Person(name: String, age: Int)

val genPerson: Gen[Person] = for
  name <- Gen.alphaStr
  age <- Gen.choose(0, 120)
yield Person(name, age)
```

Integration with MUnit

```scala
import munit.ScalaCheckSuite
import org.scalacheck.Prop.*

class MathTests extends ScalaCheckSuite:
  property("addition is commutative"):
    forAll: (a: Int, b: Int) => a + b == b + a

  property("string concatenation length"):
    forAll:
      (s1: String, s2: String) =>
      (s1 + s2).length == s1.length + s2.length
```

Common Patterns

```scala
// Test collection properties
forAll:
  (l: List[Int]) =>
  l.reverse.reverse == l  // involution

// Test algebraic laws
forAll:
  (a: Int, b: Int, c: Int) =>
  (a + b) + c == a + (b + c)  // associativity

// Round-trip testing
forAll:
  (data: MyData) =>
  decode(encode(data)) == Some(data)
```

Add in sbt build

```scala
// in build.sbt
libraryDependencies += "org.scalacheck" %% "scalacheck" % "1.18.0" % Test

// with MUnit
libraryDependencies ++= Seq(
  "org.scalameta" %% "munit" % "1.0.0" % Test,
  "org.scalameta" %% "munit-scalacheck" % "1.0.0" % Test
)
```

Key Advantages

- Automatically tests many cases, including edge cases
- Properties serve as executable specifications
- ScalaCheck shrinks failing cases to minimal examples
- Less test code to maintain than exhaustive unit tests

1.2.25 Tail recursion

A recursive function is **tail-recursive** if the recursive call is the last operation in the function.

```
import scala.annotation.tailrec
@tailrec
def factorial(n: Int, acc: Int = 1): Int =
  if n <= 1 then acc
  else factorial(n - 1, n * acc)
```

1.2.26 For-expression

```
for
  <pattern> <- <expression (iterable)>
  [if <condition>]
  yield <expression>
// equivalent to

<expression (iterable)>.filter(<condition>).map(<expression>) // Actually with withFilter
↪  instead of filter to improve performance
// Example
for
  i <- 1 until n
  if i % 3 != 0
  j <- 1 until i
  if isPrime(i + j)
yield (i, j)

// equivalent to
(1 until n)
  .withFilter(i => i % 3 != 0)
  .flatMap(i =>
    (1 until i)
      .withFilter(j => isPrime(i + j))
      .map(j => (i, j))
  )

// Other Alternative
(1 until n).withFilter: i =>
  i % 3 != 0
.flatMap: i =>
  (1 until i).withFilter: j =>
    isPrime(i + j)
  .map: j =>
    (i, j)

// With value assignation
for
  i <- 1 until n
  if i % 3 != 0
  j <- 1 until i
  sum = i + j
  if isPrime(sum)
yield (i, j, sum)
```

- A **generator** is of the form `p <- e`, where `p` is a pattern and `e` an expression whose value is a collection or monadic type.
- A **filter** is of the form `if f`, where `f` is a boolean expression.
- The sequence must start with a generator.
- If there are several generators, the last generators vary faster than the first.

- A **for-expression** is purely **syntactic sugar**: it is rewritten by the compiler into calls to `map`, `flatMap`, and `withFilter`.
- The resulting type depends on the source:
  - From a `List` => returns a `List`
  - From a `Map` => returns a `Map`
  - From a `Range` => returns an `IndexedSeq` (e.g. a `Vector`)
  - From an `Option` => returns an `Option`

Classical `for` and `while` loops Scala also supports traditional imperative loops for side-effect-based code.

```
// for loop over a range
for i <- 0 until 5 do
  println(i)

// Note that as the the same desugaring as for-expressions
(0 until 5).foreach(i => println(i))

// with multiple generators
for
  i <- 1 to 3
  j <- 1 to 2
do
  println(s"($i, $j)")

// desugared
(1 to 3).foreach(i =>
  (1 to 2).foreach(j =>
    println(s"($i, $j)")
  )
)

// with guard (filter)
for
  i <- 1 to 10
  if i % 2 == 0
do
  println(i)

// nested braces alternative
for (i <- 1 to 3; j <- 1 to 2) {
  println(s"($i, $j)")
}

// While loops are not functional programming, therefore they are really
// different constructs and not syntactic sugar.
// while loop
var i = 0
while i < 5 do
  println(i)
  i += 1

// Note: even if not automatically desugared, unfold can reproduce it
// without var and in a functional way:
LazyList.unfold(0): i =>
  if i < 5 then Some((i, i + 1))
  else None
.foreach(println)

// More complex example with accumulated state
```

```
var i2 = 0
var sum = 0
while sum < 100 do
  println(s"i=$i2, sum=$sum")
  sum += i2
  i2 += 1

// Functional equivalent with unfold
LazyList.unfold((0, 0)): (i, sum) =>
  if sum < 100 then Some(((i, sum), (i + 1, sum + i)))
  else None
.foreach: (i, sum) =>
  println(s"i=$i, sum=$sum")

// do-while loop
var j = 0
do
  println(j)
  j += 1
while j < 5

// There is no direct functional higher-order function equivalent for
// do-while loops, but an easy way is to prepend a lazy head:
(0 #:: LazyList.unfold(1): i =>
  if i < 5 then Some((i, i + 1))
  else None
).foreach(println)
```

1.2.27 Polymorphism Subtyping and Generics

Generics (Parametric polymorphism) Corresponds approximately to generics <T> in Java.

```
// Class example
sealed trait List[T]
  case class Nil[T]() extends List[T] // made Nil a class to make T known
  case class Cons[T](head: T, tail: List[T]) extends List[T]
// Some methods do not care about type
extension[T] (xs: List[T])
def length: BigInt =
  xs match
case Nil() => 0
case Cons(h, t) => 1 + t.length
def ++(ys: List[T]): List[T] =
  xs match
case Nil() => ys
case Cons(h, t) => Cons(h, t ++ ys)

// Function example
def singleton[T](x: T): List[T] = Cons(x, Nil())
// Then
singleton[Int](1)
singleton[String]("test")
// Type can often be inferred
singleton(1)
singleton("test")
```

Type erasure Type parameters do not affect evaluation in Scala. We can assume that all type parameters and type arguments are removed before evaluating the program. This is also called type erasure. Languages that use type erasure include Java, Scala, Haskell, ML, OCaml... Some other languages keep the type parameters around at run time, for example, C#, C++...

Subtyping (Inclusion polymorphism) A type S is a subtype of a type T (written $S <: T$) if every value of type S can be used in a context where a value of type T is expected. In other words, S is a more specific type than T (S extends T).

Bounds

```
def fun[T <: UpperBound](x: T): T = ... // T is a subtype of UpperBound
def fun[T >: LowerBound](x: T): T = ... // T is a supertype of LowerBound (For instance, T can
↪  be Any)
def fun[T >: LowerBound <: UpperBound](x: T): T = ... // T is between LowerBound and UpperBound
```

Variance

Liskov substitution principle: if $S <: T$ then Container$[S] <:$ Container$[T]$.

- Covariant: if $A <: B$ then $C[A] <: C[B]$ // Like Java
- Contravariant: if $A <: B$ then $C[A] >: C[B]$
- Nonvariant: if $A <: B$ then neither $C[A] <: C[B]$ nor $C[A] >: C[B]$ // Invariant is the same. Like C#

Scala lets declare the variance of a type by annotating the type parameter:

```
class C[+A]: // C is covariant
  ...
class C[-A]: // C is contravariant
  ...
class C[A]: // C is nonvariant
  ...
```

Example

```
trait Function1[-T, +R] // Contravariant in T, covariant in R
  def apply(x: T): R
```

- **T** — input argument (`x: T`); *contravariant* (`-T`): a function that accepts a more general type can replace a function that expects a more specific type.
- **R** — return value; *covariant* (`+R`): a function that returns a more specific type can replace a function that returns a more general type.

```
// If array where class then
class Array[+T] // Covariant
  // false
  def prepend(x: T): = ... // Error: cannot have a covariant type in contravariant position
  // correct
  def prepend[U >: T](x: U): Array[U] = ... // U is a supertype of T
```

Variance check The precise rules are a bit more involved, fortunately the Scala compiler performs them for us

- **covariant** type parameters can only appear in method results.
- **contravariant** type parameters can only appear in method parameters.
- **invariant** type parameters can appear anywhere.

Warning This program compiles successfully but throws a `java.lang.ArrayStoreException` at run-time. The problem is due to the **covariance of arrays**: `Array[NonEmpty]` is considered a subtype of `Array[IntSet]` at compile time, but this relationship is unsafe because arrays are mutable.

```scala
val a: Array[NonEmpty] = Array(NonEmpty(1, Empty(), Empty()))
val b: Array[IntSet] = a
b(0) = Empty()
val s: NonEmpty = a(0)
```

1.2.28 Parallelism and concurrency

Link to the topic: section 2.9.

Threads To create a new thread:

```scala
// created by extending Thread
class MyThread(val k: Int) extends Thread:
  override def run(): Unit =
    // code to be executed in the new thread

// use case
val t = new MyThread(42)
t.start() // start the thread
t.join()  // wait for the thread to finish
// note: method are called with parentheses (work without) but better to use to that method ad
↪  side effects
// .identifier and .identifier() is valid because Thread come from Java

// created by passing a Runnable to a Thread
class MyRunnable(val k: Int) extends Runnable:
  override def run(): Unit =
    // code to be executed in the new thread

// use case
val r = new MyRunnable(42)
val t = new Thread(r)
t.start() // start the thread
t.join()  // wait for the thread to finish

// created as a Virtual Thread (JVM:Java 21+)
val vt = Thread.startVirtualThread(() =>
  println(s"Running virtual thread on ${Thread.currentThread().getName}")
)
vt.join() // wait for virtual thread to finish
// virtual threads are lightweight threads managed by the JVM, not by the OS
// they support blocking operations efficiently and scale to millions of threads
```

Futures A future work almost like `Task<T>` in C#, Asyncio Coroutine and Future in Python,...

```scala
import scala.concurrent.*
import scala.concurrent.duration.*
import scala.util.*

// needed to run Futures
given ExecutionContext = ExecutionContext.global

def compute(x: Int): Future[Int] =
  Future:
    println(s"Computing $x on thread ${Thread.currentThread().getName}")
    Thread.sleep(1000)
    x * 2

@main def runFutures(): Unit =
  val f1 = compute(21)
  val f2 = compute(84)

  // combine futures
  val combined: Future[Int] =
```

```
  for
    a <- f1
    b <- f2
  yield a + b

// handle completion asynchronously
combined.onComplete:
  case Success(value) => println(s"Result = $value")
  case Failure(ex)    => println(s"Error: ${ex.getMessage}")

// wait with timeout
Await.result(combined, 3.seconds)
```

Parrallel collections Hosted at https://github.com/scala/scala-parallel-collections
To use it, add the dependency to `build.sbt`:

```
libraryDependencies ++=
  Seq("org.scala-lang.modules" %% "scala-parallel-collections" % "1.2.0")
```

Then import it:

```
import scala.collection.parallel.CollectionConverters.*

val xs = (1 to 1000000).par // Range => ParRange
val sum = xs.reduce(_ + _) // parallel sum
```

1.2.29 Collections hierarchy diagram

This diagram shows the main Scala collections and their relationships. This diagram is not exhaustive and contains simplifications for clarity.

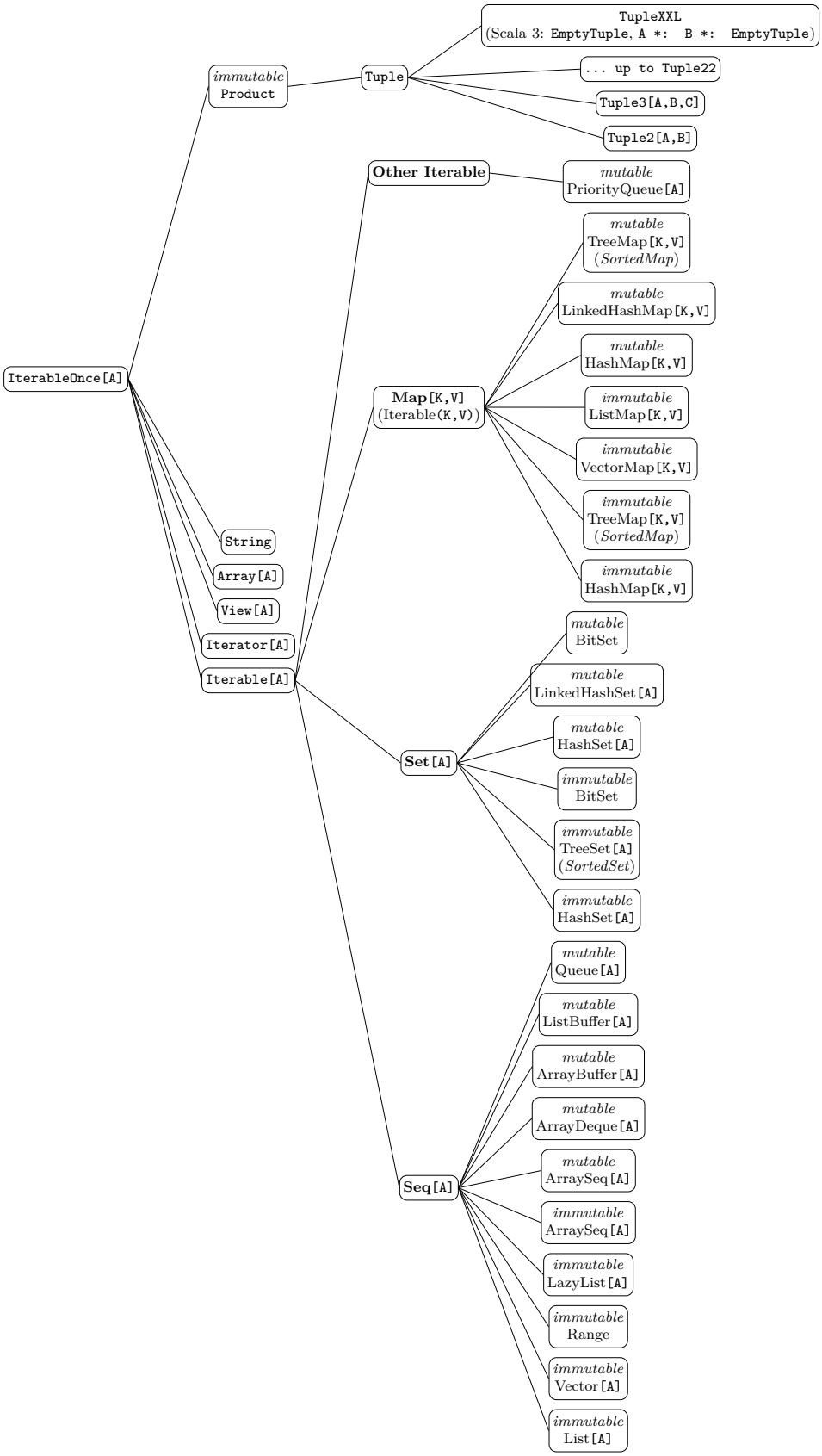


Figure 1: Scala collections hierarchy

1.2.30 Unit

```
val u: Unit = () // only value of type Unit is ()
def fun(): Unit = // function returning Unit
  println("Hello, World!")
```

1.2.31 Booleans

```
val b1: Boolean = true
val b2: Boolean = false
val b3: Boolean = b1 && b2     // logical and = conjunction
val b4: Boolean = b1 || b2     // logical or = disjunction
val b5: Boolean = !b1          // logical not = negation
val b6: Boolean = b1 ^ b2     // logical xor = exclusive or
val b7: Boolean = b1 & b2     // bitwise and
val b8: Boolean = b1 | b2     // bitwise or
```

1.2.32 Numbers

```
val i: Int = 42                     // 32-bit signed integer
val l: Long = 42000000000L          // 64-bit signed integer
val f: Float = 3.14f                // 32-bit floating point
val d: Double = 3.141592653589793   // 64-bit floating
val bd: BigDecimal = BigDecimal("1234567890.12345678901234567890") // arbitrary precision
↪  decimal
val bi: BigInt = BigInt("12345678901234567890123456789O")         // arbitrary precision integer
val c: Char = 'A'                   // 16-bit Unicode character
val s: Short = 32000               // 16-bit signed integer
val by: Byte = 127                  // 8-bit signed Integer
```

1.2.33 Tuples

```
// Define
/// Pair
val t = 1 -> 2
val t = (1, 2)

// Tuple
val t = (1, 2, 3, "test") // For 2 to 22 elements else TupleXXL
val t = 1 *: 2 *: 3 *: "test" *: EmptyTuple

val t1 = t._1 // first element
// Destructuring
val (a, b) = 1 -> 2
```

1.2.34 Strings

```
// Base
"base string"

// Raw
"""Raw string"""

// Simple interpolation
s"Formatted $identifier"
s"Formatted ${expression}"

// Raw with interpolation
raw"Raw $identifier \n not a new line" // no escape sequences but with interpolation

// Interpolation with formatting
f"Formatted $identifier%.2f" // show only 2 decimals
f"Formatted ${expression}%.5d" // show at least 5 digits, pad with 0
```

```
// Multiline
f"""Works"""
s"""Works too"""
raw"""Works as well"""

// Warning
"\t" == """\t""" // false because of raw string
s"\t" == s"""\t""" // true
f"\t" == f"""\t""" // true
raw"\t" == raw"""\t""" // true
```

1.2.35 Lists

```
val xs = List(1, 2, 3) // or 1 :: 2 :: 3 :: Nil

xs.filter(x => x % 2 == 0) // keep even elements
xs.map(x => x * x)       // square each element
xs.reduce((x, y) => x + y) // sum of elements with non-empty list and bioperator
xs.foldLeft(0)((acc, x) => acc + x) // sum of elements with initial value and bifunction
xs.foldRight(0)((acc, x) => x + acc) // same as fold left but right associative. performance may
↪  be worse
xs.sum                     // sum of elements
xs.product                 // product of elements
xs.length                  // number of elements
xs.reverse                 // reverse the list
xs ++ List(4, 5, 6)        // concatenate two lists
xs.sorted                  // sort the list
xs.last                    // last element (non-empty list)
xs.init                    // all but the last element (non-empty list)
xs.take(2)                 // first 2 elements
xs.drop(2)                 // all but the first 2 elements
xs(1)                      // second element (non-empty list)
xs.updated(1, 42)          // replace second element by 42 (non-empty list)
xs.indexOf(2)              // index of first occurrence of 2, or -1 if not found
xs.contains(2)             // true if 2 is in the list
xs.exists(x => x % 2 == 0)  // true if there is an even element
xs.forall(x => x > 0)      // true if all elements are positive
xs.mkString(", ")          // string with elements separated by ", "
xs.foreach(x => println(x)) // print each element
xs.splitAt(2)              // split into two lists at index 2
val zs = xs.zip(List("a", "b", "c")) // pair elements with another list
zs.unzip                  // unzip a list of pairs into two lists
val zs3 = xs.zip3(List("a", "b", "c"), List(true, false, true)) // pair elements with two other
↪  lists
zs3.unzip3                // unzip a list of triples into three lists // Warning : unzip4 to
↪  unzip22 do not exist
```

1.2.36 Lazy Lists

Lazy lists (previously called `Stream` in Scala 2) allow working with potentially infinite sequences by evaluating elements only when needed. Like Java streams, they are processed lazily, elements are computed on demand rather than all at once. Results are processed only when using terminal operations.

Creation

```
// Using LazyList.from for ranges
val naturals = LazyList.from(0)
val evens = LazyList.from(0, 2)  // step of 2

// Using cons operator #:: (recursive definition)
val ones: LazyList[Int] = 1 #:: ones // LazyList.cons and LazyList.empy
```

```
// Using unfold (building from a state)
val fibonacci = LazyList.unfold((0, 1)):
  case (a, b) => Some((a, (b, a + b)))
// Generates: 0, 1, 1, 2, 3, 5, 8, ...

// Using iterate (applying a function repeatedly)
val powers = LazyList.iterate(1)(_ * 2)  // 1, 2, 4, 8, 16, ...
```

Lazy operations

```
val naturals = LazyList.from(1) // Infinite LazyList

val evens = naturals.filter(_ % 2 == 0)
val squares = naturals.map(n => n * n)
val small = naturals.takeWhile(_ < 10)
val combined = evens.map(_ * 3).take(5)
```

Terminal operations

```
val firstTen = LazyList.from(1).take(10) // non-infinite LazyList

val firstTenList = firstTen.toList
firstTen.foreach(println)
val hasEven = LazyList.from(1).exists(_ % 2 == 0) // exists, forall, find
val first = LazyList.from(1).head
```

Signal Processing with lazy list A `LazyList[Short]` models an infinite audio stream evaluated on demand. It supports real-time signal processing without manual buffering or copying: past samples are cached automatically, and future samples can be computed lazily. Unlike callbacks, iterators, or fixed-size buffers, it enables smooth windowing, multiple consumers, and flexible rate control, making it ideal for continuous audio transformations.

1.2.37 Vectors

Lists are linear access. (Due to linked list structure). Vectors are created analogously to lists. Except for `xs :: x` replaced by `x +: xs` (create new vector with leading element `x` followed by all elements of `xs`) or `xs :+ x` (create new vector with trailing element `x` preceded by all elements of `xs`).

```
val xs = Vector(1, 2, 3)
val ys = 0 +: xs        // Vector(0, 1, 2, 3)
val zs = xs :+ 4        // Vector(1, 2, 3, 4)
```

1.2.38 Sequences

```
// Sequence can be created
val xs = Seq(1, 2, 3)
// just for information
// Companion object Seq.apply creates a List by default xs is List(1, 2, 3)
```

1.2.39 Ranges

```
val r1 = 1 until 10    // exclusive range
val r2 = 1 to 10       // inclusive range
val r3 = 1 to 10 by 2  // step of 2
val r4 = 10 to 1 by -1 // decreasing range
val r5 = 10 to 1 by -1 // decreasing range
val r6 = 10 until 1    // empty range
val r7 = 1 to 10 reverse // decreasing range
```

1.2.40 Option

```
// Creating options
val o1: Option[Int] = Some(42)
val o2: Option[Int] = None

// Accessing values safely
val value = o1.getOrElse(0)   // returns 42
val value2 = o2.getOrElse(0)  // returns default 0

// Mapping and chaining
val squared = o1.map(x => x * x)         // Some(1764)
val plusOne = o1.flatMap(x => Some(x+1)) // Some(43)

// Pattern matching
o1 match
  case Some(v) => println(s"Value: $v")
  case None    => println("No value")

// Filtering
val even = o1.filter(_ % 2 == 0) // Some(42)
val odd  = o1.filter(_ % 2 == 1) // None

// Combining options
val a = Some(2)
val b = Some(3)
val sum = for
  x <- a
  y <- b
yield x + y  // Some(5)

// Unsafe: throws if None
// o2.get
```

1.2.41 Try

`Try[T]` represents a computation that may either succeed with a value of type `T` or fail with an exception. It's similar to `Option`, but captures **why** the computation failed.

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

// Creating Try from computations that might throw
val t1 = Try(10 / 2)        // Success(5)
val t2 = Try(10 / 0)        // Failure(ArithmeticException: / by zero)
val t3 = Try("123".toInt)   // Success(123)
val t4 = Try("abc".toInt)   // Failure(NumberFormatException)

// Accessing values safely
val value = t1.getOrElse(0)     // 5
val value2 = t2.getOrElse(0)    // 0 (returns default on failure)

// Pattern matching
t2 match
  case Success(v) => println(s"Result: $v")
  case Failure(e) => println(s"Error: ${e.getMessage}")

// Mapping and chaining
val doubled = t1.map(_ * 2)              // Success(10)

// flatMap for chained operations
def divide(a: Int, b: Int): Try[Int] = Try(a / b)

val result = for
  x <- divide(10, 2)   // Success(5)
  y <- divide(x, 0)    // Failure (division by zero)
```

```
yield x + y            // Failure (short-circuits on first failure)

// Recovering from failures
val recovered = t2.recover:
  case _: ArithmeticException => 0 // Success(0)

// Converting to Option
val opt: Option[Int] = t1.toOption      // Some(5)
```

1.2.42 Either

`Either[E, A]` represents a value that can be one of two types: `Left[E]` for errors or `Right[A]` for success. Unlike `Try`, it allows **custom error types** instead of only exceptions.

```
// Definition
sealed trait Either[+E, +A]
case class Left[+E](value: E) extends Either[E, Nothing]
case class Right[+A](value: A) extends Either[Nothing, A]

// Creating Either values
val e1: Either[String, Int] = Right(42)
val e2: Either[String, Int] = Left("Something went wrong")

// Usage with custom error types
def divide(a: Int, b: Int): Either[String, Int] =
  if b == 0 then Left("Division by zero")
  else Right(a / b)

divide(10, 2)  // Right(5)
divide(10, 0)  // Left("Division by zero")

// Accessing values safely
val value = e1.getOrElse(0)   // 42
val value2 = e2.getOrElse(0)  // 0

// Pattern matching
e1 match
  case Right(v) => println(s"Success: $v")
  case Left(e)  => println(s"Error: $e")
```

Either is Right-Biased In Scala, `Either` is right-biased: `map`, `flatMap`, and for-comprehensions operate on the `Right` value.

```
val result: Either[String, Int] = Right(10)

result.map(_ * 2)                  // Right(20)
result.flatMap(x => Right(x + 1))  // Right(11)

// Chaining operations
def sqrt(x: Int): Either[String, Double] =
  if x < 0 then Left("Negative input")
  else Right(Math.sqrt(x))

def inverse(x: Double): Either[String, Double] =
  if x == 0 then Left("Division by zero")
  else Right(1.0 / x)

// For-comprehension (short-circuits on Left)
def compute(x: Int): Either[String, Double] =
  for
    s <- sqrt(x)
    i <- inverse(s)
  yield i
```

```
compute(4)   // Right(0.5)
compute(-1)  // Left("Negative input")
compute(0)   // Left("Division by zero")
```

Converting Between Types

```
// Option to Either
val opt: Option[Int] = Some(42)
val either: Either[String, Int] = opt.toRight("Value was None")

// Try to Either
val tryResult: Try[Int] = Try(42 / 0)
val eitherFromTry: Either[Throwable, Int] = tryResult.toEither

// Either to Option (loses error information)
val backToOption: Option[Int] = either.toOption
```

Comparison: Option vs Try vs Either

Aspect	Option[T]	Try[T]	Either[E, T]
Success case	`Some(value)`	`Success(value)`	`Right(value)`
Failure case	`None`	`Failure(exception)`	`Left(error)`
Error info	None	`Throwable`	Custom type `E`
Use when	Absence is expected	Computation may throw	Need typed errors

1.2.43 Seq vs Set vs Map

Seq

- Ordered collection of elements
- Allows duplicates

```
xs: Seq[Int] = List(1, 2, 3, 2)
xs(1) // 2
xs(2) // 3
```

Set

- Unordered collection of unique elements
- No duplicates allowed
- Efficient membership testing (fundamental operations is containing)

```
us: Set[Int] = TreeSet(1, 2, 3)
us(1) // true
us(4) // false
```

Map

- Collection of key-value pairs
- Keys are unique
- Efficient key-based access (fundamental operations are get, put)

```
kvs: Map[String, Int] = TreeMap("I" -> 1, "II" -> 2, "V" -> 5)
kvs("I") // 1
kvs.get("II") // Some(2) (Option type)
kvs.get("X") // None
kvs.getOrElse("X", 10) // 10
```

1.2.44 Useful query on collections

```
val fruits = List("apple", "banana", "orange", "kiwi", "grape", "pear", "peach", "apricot")
// sorting
fruits.sorted // List(apple, apricot, banana, grape, kiwi, orange, peach, pear)
fruits.sortWith(_.length < _.length) // List(kiwi, pear, apple, grape, peach, banana, orange,
↪  apricot)
// grouping
fruits.groupBy(_.head) // Map(a -> List(apple, apricot), b -> List(banana), g -> List(grape), k
↪  -> List(kiwi), o -> List(orange), p -> List(pear, peach))
```

1.2.45 Discouraged Features

Scala includes some features primarily for Java interoperability that are not recommended in idiomatic Scala code.

Null Null is valid, but `Option[T]` should be used instead to avoid null pointer exceptions and make absence explicit.

```
// Discouraged
val s: String = null

// Recommended
val s: Option[String] = None
```

Break and Continue Scala provides break/continue for Java developers, but functional approaches (recursion, pattern matching, filter) are preferred.

```
// Discouraged
import scala.util.control.Breaks.{break, breakable}
breakable:
  for i <- 1 to 10 do
    if i == 5 then break()
    println(i)

// Recommended alternatives
(1 to 10).takeWhile(_ != 5).foreach(println)  // using takeWhile
(1 to 10).filter(_ < 5).foreach(println)      // using filter
```

1.2.46 Contextual Abstraction

Contextual abstraction allows functions and classes to be written without knowing the exact context in which they will be called. Context can include configuration, scope, comparison methods, user credentials, security levels, etc.

The Problem Traditional approaches to handle context have drawbacks:

- **Global values:** Too rigid, no abstraction
- **Global mutable variables:** Dangerous interference between modules
- **Monkey patching:** Runtime changes to base classes are error-prone
- **Dependency injection frameworks:** Operate outside the language, harder to debug

Functional Solution: Using Clauses Scala provides `using` clauses to pass context implicitly.

```
// Without context - works only for Int
def sort(xs: List[Int]): List[Int] =
  ... if x < y then ...
```

```
// With explicit parameter - verbose
def sort[T](xs: List[T])(lessThan: (T, T) => Boolean): List[T] =
  ... if lessThan(x, y) then ...

// With using clause - elegant
def sort[T](xs: List[T])(using ord: Ordering[T]): List[T] =
  ... if ord.lt(x, y) then ...
```

Given Instances Define context values that the compiler can automatically provide:

```
// Named given instance
object Ordering:
  given Int: Ordering[Int] with
    def compare(x: Int, y: Int): Int =
      if x < y then -1 else if x > y then 1 else 0

// Anonymous given instance
given Ordering[Double] with
  def compare(x: Double, y: Double): Int = ...

// Usage - compiler infers the Ordering
val ints = List(3, 1, 2)
sort(ints)  // compiler provides Ordering.Int automatically
```

Anonymous Using Clauses When the context parameter is only passed through, you can omit its name:

```
def sort[T](xs: List[T])(using Ordering[T]): List[T] =
  ... merge(sort(fst), sort(snd)) ...  // Ordering passed implicitly

def merge[T](xs: List[T], ys: List[T])(using Ordering[T]): List[T] = ...
```

Summoning Instances Access a given instance by its type:

```
summon[Ordering[Int]]     // returns Ordering.Int
summon[Ordering[Double]]  // returns the anonymous instance

// summon is defined as:
def summon[T](using x: T): T = x
```

Given Instance Resolution The compiler searches for given instances in this order:

1. Visible instances (inherited, imported, or in enclosing scope)
2. Companion objects associated with the queried type
3. Companion objects of parent types
4. Companion objects of type arguments

Importing Given Instances Three ways to import givens:

```
// 1. By name
import scala.math.Ordering.Int

// 2. By type (preferred - most informative)
import scala.math.Ordering.{given Ordering[Int]}
import scala.math.Ordering.{given Ordering[?]}  // wildcard type

// 3. With wildcard
import scala.math.given  // all givens in scala.math
```

Priority Rules When multiple given instances match, the compiler chooses based on:

1. Closer lexical scope wins
2. Subclass definition wins over superclass
3. More specific type wins (e.g., `A[Int]` over `A[T]`)
4. Subtype wins over supertype

Syntax Reference

```
// Multiple parameters in using clause
def f(x: Int)(using a: A, b: B): Unit = ...

// Multiple using clauses
def f(x: Int)(using a: A)(using b: B): Unit = ...

// Mixed with regular parameters
def f(x: Int)(using a: A)(y: Boolean)(using b: B): Unit = ...

// Explicit using arguments (rarely needed)
f(x)(using a)(y)(using b)
```

Difference from Scala 2 Scala 3 introduces `using` and `given` keywords for clarity. Before Scala 3, implicit parameters and values were declared with the `implicit` keyword.

```
// Scala 2 (old style)
implicit val intOrdering: Ordering[Int] = new Ordering[Int] {
  def compare(x: Int, y: Int): Int = ...
}
def sort[A](list: List[A])(implicit ord: Ordering[A]): List[A] = ...

// Scala 3 (new style)
given Ordering[Int] with
  def compare(x: Int, y: Int): Int = ...
def sort[A](list: List[A])(using ord: Ordering[A]): List[A] = ...
```

1.2.47 Type Classes

A **type class** is a generic trait that defines operations for types, paired with `given` instances that implement those operations for specific types. Type classes enable *ad-hoc polymorphism*: the same operation can have different implementations for different types.

Definition Pattern A type class follows this structure:

```
// 1. Define the type class trait
trait Ordering[A]:
  def compare(x: A, y: A): Int

// 2. Provide given instances for specific types
object Ordering:
  given Int: Ordering[Int] with
    def compare(x: Int, y: Int) =
      if x < y then -1 else if x > y then 1 else 0

  given String: Ordering[String] with
    def compare(x: String, y: String) = x.compareTo(y)
```

Ad-Hoc Polymorphism Type classes provide a form of polymorphism where the same method works for any type that has a corresponding given instance:

```
def sort[T](xs: List[T])(using Ordering[T]): List[T] = ...

// At compile-time, the compiler resolves the specific implementation
sort(List(3, 1, 2))           // Uses Ordering.Int
sort(List("c", "a", "b"))     // Uses Ordering.String
sort(List(List(1), List(2)))  // Uses listOrdering(using Ordering.Int)
```

This is called *ad-hoc polymorphism* because `Ordering[A]` has different implementations for different types `A`.

Context Bounds The `(using TypeClass[T])` pattern is so common that Scala provides a shorthand called **context bounds**:

```
// Verbose form
def sort[T](xs: List[T])(using Ordering[T]): List[T] = ...

// Context bound (equivalent)
def sort[T: Ordering](xs: List[T]): List[T] = ...
```

Read `T: Ordering` as "`T` has an `Ordering`".

Translation Rule A method with multiple context bounds:

```
def f[T: U1 : U2 : U3](ps): R = ...
// expands to:
def f[T](ps)(using U1[T], U2[T], U3[T]): R = ...
```

Named Context Bounds (Scala 3.6+) When you need to access the instance directly, use the `as` syntax:

```
// Without named bound (requires summon)
def reduce[T: Monoid](xs: List[T]): T =
  xs.foldLeft(summon[Monoid[T]].unit)(_.combine(_))

// With named bound (Scala 3.6+)
def reduce[T: Monoid as m](xs: List[T]): T =
  xs.foldLeft(m.unit)(_.combine(_))
```

Conditional Instances Given instances can depend on other givens. Context bounds also work in given definitions:

```
// List ordering depends on element ordering
given listOrdering[T: Ordering]: Ordering[List[T]] with
  def compare(xs: List[T], ys: List[T]): Int = (xs, ys) match
    case (Nil, Nil) => 0
    case (Nil, _) => -1
    case (_, Nil) => 1
    case (x :: xs1, y :: ys1) =>
      val c = summon[Ordering[T]].compare(x, y)
      if c != 0 then c else compare(xs1, ys1)

// Pair ordering
given pairOrdering[A: Ordering, B: Ordering]: Ordering[(A, B)] with
  def compare(x: (A, B), y: (A, B)): Int =
    val c = summon[Ordering[A]].compare(x._1, y._1)
    if c != 0 then c else summon[Ordering[B]].compare(x._2, y._2)

// Recursive resolution
val xss: List[List[Int]] = ...
sort(xss)  // Compiler infers: listOrdering(using Ordering.Int)
```

Extension Methods in Type Classes Type class traits commonly define extension methods that become available whenever a given instance is in scope:

```
trait Ordering[A]:
  def compare(x: A, y: A): Int

  extension (x: A)
    def < (y: A): Boolean = compare(x, y) < 0
    def <= (y: A): Boolean = compare(x, y) <= 0
    def > (y: A): Boolean = compare(x, y) > 0
    def >= (y: A): Boolean = compare(x, y) >= 0

// Usage: extension methods are visible when Ordering[T] is available
def merge[T: Ordering](xs: List[T], ys: List[T]): List[T] = (xs, ys) match
  case (Nil, _) => ys
  case (_, Nil) => xs
  case (x :: xs1, y :: ys1) =>
    if x < y then x :: merge(xs1, ys)  // Uses < extension method
    else y :: merge(xs, ys1)
```

Retroactive Extension Type classes support **retroactive extension**: adding capabilities to existing types without modifying their original definitions.

```
// Original type (cannot be modified)
case class Rational(numer: Int, denom: Int)

// Add ordering capability later, without changing Rational
given Ordering[Rational] with
  def compare(x: Rational, y: Rational) =
    Ordering.Int.compare(x.numer * y.denom, y.numer * x.denom)

// Now Rational can be sorted
val rationals = List(Rational(1, 2), Rational(1, 3), Rational(2, 3))
sort(rationals)  // Works!
```

Caveat: Retroactive extensions must be explicitly imported or defined where used, since they cannot be placed in companion objects of types you don't control.

Type Classes in Other Languages

Language	Name
Haskell	Type class (original source of the name)
Rust	`trait` (Scala traits ≈ `impl Trait` or `dyn Trait`)
Swift	`protocol`
Lean 4	Type class

Type Classes vs. Subtype Polymorphism

Aspect	Subtype Polymorphism	Type Classes
When defined	At class definition	Anytime (retroactive)
Conditional	No	Yes (can depend on other instances)
Multiple impls	No (one per type)	Yes (explicit import)
Discovery	Automatic	Via companion objects or imports

1.2.48 Abstract Algebra with Type Classes

Type classes naturally model algebraic structures, allowing generic algorithms over any type that satisfies algebraic properties.

SemiGroup A **semigroup** has an associative binary operation:

```
trait SemiGroup[T]:
  extension (x: T) def combine(y: T): T

// Generic reduction for any semigroup
def reduce[T: SemiGroup](xs: List[T]): T =
  xs.reduceLeft(_.combine(_))
```

Monoid A **monoid** is a semigroup with an identity element (unit):

```
trait Monoid[T] extends SemiGroup[T]:
  def unit: T

// Reduce that handles empty lists
def reduce[T](xs: List[T])(using m: Monoid[T]): T =
  xs.foldLeft(m.unit)(_.combine(_))

// Or with context bound and summon
def reduce[T: Monoid](xs: List[T]): T =
  xs.foldLeft(summon[Monoid[T]].unit)(_.combine(_))
```

Multiple Instances A type can be a monoid in multiple ways. For example, `Int` has two natural monoid structures:

```
// Addition monoid
given sumMonoid: Monoid[Int] with
  extension (x: Int) def combine(y: Int): Int = x + y
  def unit: Int = 0

// Multiplication monoid
given prodMonoid: Monoid[Int] with
  extension (x: Int) def combine(y: Int): Int = x * y
  def unit: Int = 1

// Must explicitly import the desired instance
import sumMonoid.given
reduce(List(1, 2, 3))  // 6 (using addition)
```

Type Class Laws Algebraic type classes are defined not just by signatures, but by **laws** that instances must satisfy.

For `Monoid[T]`, these laws must hold for all `x, y, z: T`:

1. **Associativity:** `x.combine(y).combine(z) == x.combine(y.combine(z))`
2. **Left identity:** `unit.combine(x) == x`
3. **Right identity:** `x.combine(unit) == x`

Laws can be verified through formal proofs or property-based testing with ScalaCheck:

```
import org.scalacheck.Prop.forAll

// Test associativity law
forAll { (x: Int, y: Int, z: Int) =>
  x.combine(y).combine(z) == x.combine(y.combine(z))
}

// Test identity laws
forAll { (x: Int) =>
  (sumMonoid.unit.combine(x) == x) && (x.combine(sumMonoid.unit) == x)
}
```

1.2.49 Context Passing

Beyond type classes, givens are used for **context passing**: implicitly threading values through a computation without explicit parameters.

Two Uses of Givens

- **Type classes:** What is the definition of `TC[A]` for a type class trait `TC` and type argument `A`?
- **Context passing:** What is the currently valid definition of type `T`?

Use Cases Context passing is useful for propagating:

- Current configuration or settings
- Available capabilities or permissions
- Security levels or credentials
- Layout schemes for rendering
- User identity or access control

Case Study: Conference Management Consider a system where paper scores must be hidden from authors:

```
case class Person(name: String)
case class Paper(title: String, authors: List[Person], body: String)

object ConfManagement:
  type Viewers = Set[Person]

  class Conference(ratings: (Paper, Int)*):
    private val realScore = ratings.toMap

    def papers: List[Paper] = ratings.map(_._1).toList

    def score(paper: Paper, viewers: Viewers): Int =
      if paper.authors.exists(viewers.contains) then -100
      else realScore(paper)

    def rankings(viewers: Viewers): List[Paper] =
      papers.sortBy(score(_, viewers)).reverse

    def ask[T](p: Person, query: Viewers => T): T =
      query(Set(p))

    def delegateTo[T](p: Person, query: Viewers => T)(viewers: Viewers): T =
      query(viewers + p)
```

Problem: Tedious Parameter Passing Every function must explicitly pass `viewers`:

```
conf.rankings(viewers).takeWhile(conf.score(_, viewers) > 80)
```

Tamper-Proofing with Opaque Types Prevent bypassing the access control by making `Viewers` opaque:

```
object ConfManagement:
  opaque type Viewers = Set[Person]

  // Inside ConfManagement: Viewers = Set[Person] (equality known)
  // Outside: Viewers is abstract (cannot create instances)
```

Opaque type aliases hide the underlying type outside their defining scope. Since `Viewers` appears abstract externally, code cannot create fake `Viewers` values—the only way to obtain one is through the system's API.

Using Clauses for Automatic Passing Replace explicit parameters with `using` clauses:

```
class Conference(ratings: (Paper, Int)*):
  def score(paper: Paper)(using viewers: Viewers): Int =
    if paper.authors.exists(viewers.contains) then -100
    else realScore(paper)

  def rankings(using viewers: Viewers): List[Paper] =
    papers.sortBy(score(_)).reverse  // viewers passed implicitly

  def delegateTo[T](p: Person, query: Viewers => T)(using viewers: Viewers): T =
    query(viewers + p)

// Usage is cleaner
conf.rankings.takeWhile(conf.score(_) > 80)
```

Benefits of Opaque Types with Givens

1. **No accidental connections:** Since `Viewers` is abstract, it won't match given instances of other types like `Set[Person]`.
2. **Enforced correctness:** Only one `Viewers` value exists in scope (from the query parameter).
3. **Clean code:** No explicit passing needed.

Best Practice: Specific Types Never use common types for globally visible givens:

```
// TERRIBLE - will cause chaos
given Int = 1
def f(x: Int)(using delta: Int) = x + delta

// GOOD - use specific or opaque types
opaque type Delta = Int
given defaultDelta: Delta = 1
def f(x: Int)(using delta: Delta) = x + delta
```

1.2.50 Context Function Types

Note: This section covers additional material not required for the course.

Context function types eliminate the need for explicit `using` clauses in method signatures.

Context Function Syntax A context function uses `?=>` instead of `=>`:

```
// Regular function type
val f: A => B

// Context function type (parameter is implicit)
val g: A ?=> B
```

Creating Context Functions Use `?=>` in lambda syntax:

```
def rankings = (viewers: Viewers) ?=>
  papers.sortBy(score(_, viewers)).reverse
```

The type of `rankings` is `Viewers ?=> List[Paper]`.

Typing Rules

1. **Automatic argument inference:** When a context function is applied, arguments are inferred:

```
val f: A ?=> B = ...
given a: A = ...
f  // Expands to f(using a)
```

2. **Automatic creation:** If expected type is `A ?=> B`, an expression `b` expands to `(_: A) ?=> b`:

```
val f: Int ?=> String = "value"  // Becomes (_: Int) ?=> "value"
```

Type Alias Pattern Define a type alias for cleaner signatures:

```
type Viewed[T] = Viewers ?=> T

// Replace (using Viewers): SomeType with : Viewed[SomeType]
def score(paper: Paper): Viewed[Int] = ...
def rankings: Viewed[List[Paper]] = ...
def delegateTo[T](p: Person, query: Viewed[T]): Viewed[T] = ...
```

Trade Types for Parameters Context function types take implicit parameters one step further:

- **Using clauses:** Developer writes required type, compiler infers the term.
- **Context functions:** Developer writes return type, compiler infers the parameter.

1.2.51 Summary: Contextual Abstraction

Concept	Key Point
`using` clause	Implicit parameter passed by compiler
`given` instance	Value automatically provided for a type
`summon[T]`	Retrieve a given instance by type
Type class	Generic trait + given instances for ad-hoc polymorphism
Context bound	`[T: TC]` shorthand for `(using TC[T])`
Retroactive extension	Add capabilities without modifying original types
Type class laws	Mathematical properties instances must satisfy
Context passing	Using givens for configuration, security, etc.
Opaque types	Hide implementation to prevent tampering
Context functions	`A ?=> B` — implicit parameters without `using` clauses

1.2.52 Generators

A generator `Gen[T]` produces random values of type `T` for property-based testing.

Built-in Generators ScalaCheck provides generators for common types:

```
import org.scalacheck.Gen
import org.scalacheck.Arbitrary.arbitrary

// Basic types (automatic)
arbitrary[Int]      // Random integers
arbitrary[String]   // Random strings
arbitrary[Boolean]  // Random booleans
arbitrary[List[A]]  // Random lists

// Generate from a range
val smallInt: Gen[Int] = Gen.choose(0, 100)
```

```
val letter: Gen[Char] = Gen.choose('a', 'z')

// Generate from specific values
val diceRoll: Gen[Int] = Gen.oneOf(1, 2, 3, 4, 5, 6)
val color: Gen[String] = Gen.oneOf("red", "green", "blue")

// Constant generator
val alwaysZero: Gen[Int] = Gen.const(0)
```

Generator Combinators Combine generators to create complex structures:

```
// Lists with specific size
val threeInts: Gen[List[Int]] = Gen.listOfN(3, arbitrary[Int])

// Optional values
val maybeInt: Gen[Option[Int]] = Gen.option(arbitrary[Int])

// Frequency-weighted generation
val biasedCoin: Gen[String] = Gen.frequency(
  (7, Gen.const("heads")),  // 70% probability
  (3, Gen.const("tails"))   // 30% probability
)

// Filtered generation
val evenInt: Gen[Int] = arbitrary[Int].suchThat(_ % 2 == 0)
val positiveInt: Gen[Int] = arbitrary[Int].suchThat(_ > 0)
```

Custom Generators with For-Comprehensions Build generators using for-comprehensions:

```
case class Person(name: String, age: Int)

val genPerson: Gen[Person] = for
  name <- Gen.alphaStr                    // Random alphabetic string
  age <- Gen.choose(0, 120)               // Age between 0 and 120
yield Person(name, age)

// More complex example
case class Email(user: String, domain: String)

val genEmail: Gen[Email] = for
  user <- Gen.alphaLowerStr.suchThat(_.nonEmpty)
  domain <- Gen.oneOf("gmail.com", "epfl.ch", "example.org")
yield Email(user, domain)

// Using the generator in properties
forAll(genPerson):
  person =>
  person.age >= 0 && person.age <= 120
```

Recursive Generators Generate recursive data structures with size control:

```
sealed trait Tree[+A]
case class Leaf[A](value: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]

def genTree[A](genA: Gen[A]): Gen[Tree[A]] =
  Gen.sized:
    size =>
    if size <= 0 then
      genA.map(Leaf(_))
    else
      Gen.oneOf(
        genA.map(Leaf(_)),
        for
```

```
          left <- Gen.resize(size / 2, genTree(genA))
          right <- Gen.resize(size / 2, genTree(genA))
        yield Branch(left, right)
      )

// Use with property
forAll(genTree(arbitrary[Int])): tree =>
  countLeaves(tree) > 0
```

Implicit Arbitrary Instances Make generators implicit for automatic use:

```
import org.scalacheck.Arbitrary

case class Point(x: Int, y: Int)

given Arbitrary[Point] = Arbitrary:
  for
    x <- Gen.choose(-100, 100)
    y <- Gen.choose(-100, 100)
  yield Point(x, y)

// Now Point is automatically generated in forAll
forAll:
  (p: Point) =>
  p.x >= -100 && p.x <= 100
```

Generator Methods Useful methods for transforming generators:

```
val gen: Gen[Int] = Gen.choose(1, 10)

// Transform values
val doubled: Gen[Int] = gen.map(_ * 2)

// Flat mapping for dependent generation
val pair: Gen[(Int, Int)] = gen.flatMap:
  x =>
  Gen.choose(x, x + 10).map(y => (x, y))

// Filtering (use sparingly - can be slow)
val evenGen: Gen[Int] = gen.suchThat(_ % 2 == 0)

// Retry if filter fails too often
val safeEven: Gen[Int] = gen.retryUntil(_ % 2 == 0)

// Sample generator values (for debugging)
gen.sample  // Option[Int]: Some random value
```

Common Generator Patterns

```
// Non-empty collections
val nonEmptyList: Gen[List[Int]] =
  Gen.nonEmptyListOf(arbitrary[Int])

// Containers with size bounds
val boundedList: Gen[List[Int]] =
  Gen.listOfN(Gen.choose(1, 10).sample.get, arbitrary[Int])

// Pairs and tuples
val pair: Gen[(Int, String)] = for
  i <- arbitrary[Int]
  s <- arbitrary[String]
yield (i, s)

// Maps with specific keys
```

```
val scoreMap: Gen[Map[String, Int]] =
  Gen.mapOfN(5,
    Gen.zip(Gen.alphaStr, Gen.choose(0, 100))
  )
```

Testing Generators Verify generators produce expected distributions:

```
import org.scalacheck.Prop.{forAll, classify}

property("generator distribution") = forAll(Gen.choose(1, 100)):
  n =>
  classify(n < 25, "low"):
    classify(n >= 25 && n < 75, "medium"):
      classify(n >= 75, "high"):
        n >= 1 && n <= 100
```

section

Best Practices

- Use `Gen.choose` for bounded numeric ranges
- Avoid excessive `suchThat` filtering (can cause timeouts)
- Use `Gen.sized` for recursive structures to control depth
- Make generators implicit via `Arbitrary` for cleaner tests
- Test your generators to ensure proper distribution

1.3 Patterns and Functional Design

1.3.1 State Machines

A state machine models computation as transitions between discrete states. In functional programming, state machines are implemented using:

- Immutable states (enums or case classes)
- Pure transition functions
- Explicit state threading

Basic Pattern The core idea: a state machine is just a function that takes a current state and an input, and returns a new state.

```
// 1. Define states
enum State:
  case StateA, StateB, StateC

// 2. Define inputs (events)
enum Input:
  case Event1, Event2

// 3. Define transition function
def transition(state: State, input: Input): State =
  (state, input) match
    case (State.StateA, Input.Event1) => State.StateB
    case (State.StateB, Input.Event2) => State.StateC
    case (s, _) => s  // default: stay in current state

// 4. Run the state machine
val s0 = State.StateA
```

```
val s1 = transition(s0, Input.Event1)  // StateB
val s2 = transition(s1, Input.Event2)  // StateC
```

Example: Traffic Light

```
enum Light:
  case Red, Yellow, Green

enum Event:
  case Timer

def nextLight(current: Light, event: Event): Light =
  (current, event) match
    case (Light.Red, Event.Timer) => Light.Green
    case (Light.Green, Event.Timer) => Light.Yellow
    case (Light.Yellow, Event.Timer) => Light.Red

// Process multiple events
def runLights(initial: Light, events: List[Event]): Light =
  events.foldLeft(initial)(nextLight)

// Usage
val sequence = List.fill(5)(Event.Timer)
runLights(Light.Red, sequence)  // Green (after 5 transitions)
```

With Output Sometimes we need to produce output during transitions:

```
enum DoorState:
  case Locked, Unlocked

enum Action:
  case InsertCoin, Push

enum Message:
  case Click, Beep, Nothing

// Transition returns (new state, output)
def door(state: DoorState, action: Action): (DoorState, Message) =
  (state, action) match
    case (DoorState.Locked, Action.InsertCoin) =>
      (DoorState.Unlocked, Message.Click)
    case (DoorState.Locked, Action.Push) =>
      (DoorState.Locked, Message.Beep)
    case (DoorState.Unlocked, Action.Push) =>
      (DoorState.Locked, Message.Click)
    case (s, _) =>
      (s, Message.Nothing)
```

Key insight: State machines are just data + pure functions. No mutation needed.

1.3.2 Mutation and Functional Programming

While Loops While loops allow repeated execution based on a condition:

```
def multiply(x: BigInt, y: BigInt): BigInt =
  require(y >= 0)
  var bound = y
  var result: BigInt = 0
  while bound > 0 do
    result = result + x
    bound -= 1
  result
.ensuring(_ == x * y)
```

Implementing While as a Function The `while` construct can be defined as a tail-recursive function:

```
def whileDo(condition: => Boolean)(command: => Unit): Unit =
  if condition then
    command
    whileDo(condition)(command)
```

Parameters must be passed by name (`=>`) for re-evaluation in each iteration.

For Loops Without Yield For loops without `yield` execute side effects:

```
for i <- 1 until 3 do
  System.out.print(s"$i ")  // prints: 1 2

// Nested loops
for
  i <- 1 until 3
  j <- "abc"
do
  println(s"$i $j")
```

These desugar to `foreach` calls:

```
(1 until 3).foreach(i => System.out.print(s"$i "))
```

Mutable Collections Scala provides mutable collections in `scala.collection.mutable`:

```
import scala.collection.mutable.*

val buffer: ListBuffer[Int] = ListBuffer(3)
buffer.addOne(42)  // buffer now contains 3, 42

// Copying vs mutating concatenation
val lst1 = ListBuffer(1, 2, 3)
val lst2 = ListBuffer(10, 20)
val lst3 = lst1 ++ lst2      // creates copy
val lst4 = lst1 ++= lst2      // mutates lst1 in place
```

Arrays Arrays provide efficient mutable sequences with O(1) access:

```
val a = Array(10, 20, 30)
val b = Array.fill(5)(42)              // Array(42,42,42,42,42)
val c = Array.tabulate(5)(i => 10*i)  // Array(0,10,20,30,40)

a(0) = 15  // mutation
```

Classes with Mutable State

```
case class Accumulator(private var sum: BigInt):
  def get = sum
  def add(x: BigInt): Unit =
    sum = sum + x

val acc = Accumulator(0)
acc.add(100)
acc.add(30)
acc.get  // 130
```

Side Effects Break Referential Transparency In pure functional programming, `x == x` always holds. Mutation breaks this property:

```
case class Counter(var current: Long):
  def next: Long =
    current += 1
    current

val c = Counter(0)
assert(c.next == c.next)  // FAILS: evaluates to 1 == 2
```

The expression expands to:

```
val v1 = c.next  // returns 1, counter becomes 1
val v2 = c.next  // returns 2, counter becomes 2
assert(v1 == v2) // false
```

Example: Non-Deterministic Function

```
def f(x: Long): Long =
  if c.next > 10 then x else x + 100

// f(5) sometimes returns 5, sometimes 105
// Result depends on counter state
```

Recovering Mathematical Reasoning Two approaches restore predictable behavior:

Approach 1: Explicit State Threading Convert side effects into explicit parameters and return values:

```
// Impure: reads and modifies c.current
def next: Long =
  current += 1
  current

// Pure: explicit state in/out
def next_pure(current: Long): (Long, Long) =
  (current + 1, current + 1)  // (new counter, result)
```

Example: Tree renaming

```
sealed abstract class Tree
case class Leaf(s: String) extends Tree
case class Node(left: Tree, right: Tree) extends Tree

// Impure version (uses mutable counter)
extension (t: Tree)
  def distVersion: Tree =
    t match
      case Leaf(s) => Leaf(s + "_" + c.next.toString)
      case Node(left, right) => Node(left.distVersion, right.distVersion)

// Pure version (explicit counter threading)
extension (t: Tree)
  def distVersion_pure(counter: Long): (Tree, Long) =
    t match
      case Leaf(s) =>
        val (res, c1) = next_pure(counter)
        (Leaf(s + "_" + res.toString), c1)
      case Node(left, right) =>
        val (left1, c1) = left.distVersion_pure(counter)
        val (right1, c2) = right.distVersion_pure(c1)
        (Node(left1, right1), c2)
```

Approach 2: Hoare Logic Hoare logic uses triples `{pre} body {post}` to reason about imperative code:

```
def multiply(x: BigInt, y: BigInt): BigInt =
  require(y >= 0)
  var bound = y
  var result: BigInt = 0
  assert(bound == y && result == 0)
  while bound > 0 do
    assert(result == (y - bound)*x && bound > 0)  // loop invariant
    result = result + x
    bound -= 1
    assert(result == (y - bound)*x && bound >= 0)
  assert(result == y*x && bound == 0)
  result
.ensuring(_ == x * y)
```

The loop invariant `result == (y - bound) * x` holds before and after each iteration.

Key Hoare Logic Rules:

Assignment:

$$\{\, p(e) \,\}\ x = e\ \{\, p(x) \,\}$$

While loop:

$$\frac{\{\, p \wedge \text{cond} \,\}\ \text{body}\ \{\, p \,\}}{\{\, p \,\}\ \texttt{while cond do body}\ \{\, p \wedge \neg\text{cond} \,\}}$$

State Machine Implementations Compared

Functional (Immutable)

```
case class StateF(flipped: Vector[Boolean]):
  def click(i: Int): StateF =
    StateF(flipped.updated(i, !flipped(i)))

val s0 = StateF(Vector(false, false, false, false))
val s1 = s0.click(1)
val s2 = s1.click(2)
// All versions (s0, s1, s2) remain accessible
// Copy: O(log n), Access: O(log n)
```

Shallow Mutation

```
case class StateSh(var flipped: Vector[Boolean]):
  def click(i: Int): Unit =
    flipped = flipped.updated(i, !flipped(i))

val s = StateSh(Vector(false, false, false, false))
s.click(1)
s.click(2)
// Only current state accessible
// Copy: O(log n), Access: O(log n)
```

Deep Mutation

```
case class StateD(flipped: Array[Boolean]):
  def click(i: Int): Unit =
    flipped(i) = !flipped(i)

val s = StateD(Array(false, false, false, false))
s.click(1)
s.click(2)
// No copying, direct mutation
// Copy: O(1), Access: O(1)
// Trade-off: cannot keep old versions without full array copy
```

Aliasing Problems Multiple references to the same mutable object cause unexpected behavior:

```
extension (a1: Array[Int])
  def +(a2: Array[Int]): Array[Int] =
    val result = a1  // aliasing!
    (0 until result.length).foreach: i =>
      result(i) = a1(i) + a2(i)
    result

val a1 = Array(100, 100, 100)
val a2 = Array(5, 5, 5)
val example = a1 + a2 + a1  // Array(210, 210, 210) - not 205!
// First +: a1 becomes Array(105,105,105)
// Second +: a1 already modified, so 105+105=210
```

Guidelines **Prefer immutability for:**

- Correctness and reasoning
- Multiple state versions
- Concurrent/parallel code
- Maintainable code

Consider mutation for:

- Performance-critical code
- Imperative library interfaces
- Low-level data structures
- Localized state changes

Best practice: Encapsulate mutation in small modules with pure interfaces.

1.3.3 Functional Interfaces with Imperative Implementations

The strategy is to implement a pure functional interface using internal mutation for efficiency, while hiding implementation details from external code.

Goal Replace a pure but inefficient function `f` with an optimized version `f'` such that:

1. `f'` is more efficient than `f`
2. `f'` uses mutation internally for performance
3. `f'(x)` returns the same value as `f(x)` for all `x`

If the implementation is correct, replacing `f` with `f'` preserves program behavior while improving performance.

1.3.4 Caching Patterns

Lazy Values Review Scala provides lazy evaluation with `lazy val`:

```
// Function: evaluated every time
val x = () =>
  println("Evaluating x")
  42
x()  // Evaluating x
x()  // Evaluating x (again)
```

```
// Lazy val: evaluated once
lazy val y =
  println("Evaluating y")
  42
y    // Evaluating y
y    // (no output, cached result)
```

LazyCell Class A `LazyCell` encapsulates lazy evaluation:

```
class LazyCell[+A](init: => A):
  lazy val get = init

val lc = LazyCell({println("Computing"); 42})
lc.get  // Computing
        // 42
lc.get  // 42 (no recomputation)
```

LazyList Structure Scala's `LazyList` uses `LazyCell` for the tail:

```
type LazyList[+A] = LazyCell[ListState[A]]

trait ListState[+A]
object Empty extends ListState[Nothing]
case class Cons[+A](head: A, tail: LazyList[A]) extends ListState[A]
```

The tail is lazy, allowing infinite sequences without stack overflow.

LazyCell with Mutation Implementation using internal mutation for efficiency:

```
class LazyCell[+A](val init: () => A):
  private var cached: Option[A] = None

  def get: A =
    cached match
      case Some(a) => a
      case None =>
        cached = Some(init())
        cached.get
```

This implements the pure interface:

```
class LazyCell[+A](val init: () => A):
  def get: A = init()
```

Correctness: Object Invariant The `LazyCell` maintains an invariant ensuring correctness:

```
class LazyCell[+A](val init: () => A):
  private var cached: Option[A] = None

  def valid: Boolean =
    cached == None || cached == Some(init())  // invariant

  def get: A =
    require(valid)
    cached match
      case Some(a) => a
      case None =>
        cached = Some(init())
        cached.get
  .ensuring(res => valid && res == init())
```

Invariant: `cached == None || cached == Some(init())`

Proof sketch (induction on execution steps):

- Initially: `cached == None` (constructor)
- If a step doesn't modify `cached`, invariant holds
- If a step modifies `cached`, it must be in `get` (private field)
- In `get`: `cached` becomes `Some(init())`, preserving invariant

Cached Function (Memoization) Generalize `LazyCell` to cache function results:

```
case class CachedFunction[-A, +B](val f: A => B):
  private var cache: Map[A, B] = Map()

  def apply(a: A): B =
    cache.get(a) match
      case Some(b) =>
        println(s"Cache hit: $a -> $b")
        b
      case None =>
        val b = f(a)
        cache = cache.updated(a, b)
        b

val csin = CachedFunction(math.sin)
csin(0.4)  // 0.3894183423086505
csin(0.4)  // Cache hit: 0.4 -> 0.3894183423086505
```

Invariant

```
def valid: Boolean =
  cache.keys.forall(a => cache.get(a) == Some(f(a)))
```

All cached values must equal `f(a)` for their key `a`.

Aliasing Risk Exposing mutable state breaks correctness:

```
case class CachedFunction[-A, +B](val f: A => B):
  private var cache: Map[A, B] = Map()
  def getCache: Map[A, B] = cache  // BREAKS CORRECTNESS!
```

External code could modify the cache, violating the invariant.

1.3.5 Memoization for Recursive Functions

Fibonacci Example Naive recursive Fibonacci has exponential complexity:

```
def fib(n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else fib(n - 1) + fib(n - 2)

// Complexity: O(fib(n)) >= O(2^(n/2))
```

Applying `CachedFunction` to `fib` doesn't help:

```
val cf = CachedFunction(fib)
cf(44)  // Still slow! Only caches final result, not intermediate calls
```

Memoizing Recursive Calls Cache intermediate results by intercepting recursive calls:

```
var cache: scala.collection.mutable.Map[Int, Int] =
  scala.collection.mutable.Map()

def fib(n: Int): Int =
```

```
  if n == 0 then 0
  else if n == 1 then 1
  else memo_fib(n - 1) + memo_fib(n - 2)

def memo_fib(a: Int): Int =
  cache.get(a) match
    case Some(b) => b
    case None =>
      val b = fib(a)
      cache(a) = b
      b

// Now O(n) time complexity
```

Abstracting Recursion Separate the recursive structure from the memoization logic:

```
// Recursor: parameterizes recursive calls
def fibR(rec: Int => Int, n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else rec(n - 1) + rec(n - 2)

// Generic memoization
def memo(H: (Int => Int, Int) => Int): Int => Int =
  val cache: scala.collection.mutable.Map[Int, Int] =
    scala.collection.mutable.Map()

  def rec(a: Int): Int =
    cache.get(a) match
      case Some(b) => b
      case None =>
        val b = H(rec, a)
        cache(a) = b
        b
  rec

// Combine them
def fib(x: Int) = memo(fibR)(x)
```

Generic Memoization Generalize to any types:

```
def memo[A, B](H: (A => B, A) => B): A => B =
  val cache: scala.collection.mutable.Map[A, B] =
    scala.collection.mutable.Map()

  def rec(a: A): B =
    cache.get(a) match
      case Some(b) => b
      case None =>
        val b = H(rec, a)
        cache(a) = b
        b
  rec
```

`memo` takes a recursor `H` and creates a memoized function `f` such that `H(f, x) == f(x)` for all `x`.

1.3.6 Dynamic Programming

Concept Dynamic programming solves problems bottom-up, computing smaller subproblems first:

- Memoization: top-down (compute on demand, cache results)

- Dynamic programming: bottom-up (compute in order, store in array/table)

Advantages:

- No cache lookup overhead
- Predictable memory usage
- Often uses arrays instead of maps

Example: Floyd-Warshall Algorithm Find shortest paths between all pairs of vertices in a weighted graph.

Problem: Given directed graph with distances `d(from, to)`, find shortest path between every pair of nodes.

Recursive definition:

```
// path(from, to, k): shortest path using only nodes 0..k-1 as intermediates
def path(from: Int, to: Int, k: Int): Int =
  if k == 0 then d(from, to)
  else min(
    path(from, to, k - 1),              // don't use node k
    path(from, k, k - 1) + path(k, to, k - 1)  // go through k
  )
```

Complexity: Exponential (3 recursive calls).

Memoization approach: Store results in $O(N^3)$ table.

Dynamic programming approach: Use only $O(N^2)$ space by computing layer by layer:

```
def floydWarshall(d: Array[Array[Int]]): Array[Array[Int]] =
  val N = d.length
  var p = d.map(_.clone())
  var k = 0

  while k < N do
    p = updateDistances(p, k)
    k += 1
  p

def updateDistances(p: Array[Array[Int]], k: Int): Array[Array[Int]] =
  val N = p.length
  val newP = p.map(_.clone())
  var from = 0

  while from < N do
    var to = 0
    while to < N do
      newP(from)(to) = min(p(from)(to), p(from)(k) + p(k)(to))
      to += 1
    from += 1
  newP
```

Time: $O(N^3)$, **Space:** $O(N^2)$.

1.3.7 Exceptions

Problem with Partial Functions Some functions are undefined for certain inputs:

```
def recip100(v: Int): Int =
  100 / v

def f(x: Int, y: Int): Int =
```

```
  recip100(x) + recip100(y)

// f(0, 5) crashes with ArithmeticException
```

Solution 1: Option Type

```
def recip100(v: Int): Option[Int] =
  if v == 0 then None
  else Some(100 / v)

def f(x: Int, y: Int): Option[Int] =
  recip100(x) match
    case None => None
    case Some(vx) =>
      recip100(y) match
        case None => None
        case Some(vy) => Some(vx + vy)
```

Drawback: Verbose, nested pattern matching.

Solution 2: Exceptions

```
class ReciprocalOfZero extends Exception

def recip100(v: Int): Int =
  if v == 0 then throw new ReciprocalOfZero
  else 100 / v

def f(x: Int, y: Int): Int =
  recip100(x) + recip100(y)

// Caller handles exception:
try
  f(0, 5)
catch
  case _: ReciprocalOfZero => println("Division by zero")
```

Advantage: Concise code, error handling separate.

Exception Evaluation Rules Expressions evaluate to success `S(value)` or failure `F(exception)`:

$$\begin{aligned} \texttt{throw e} &\Rightarrow F(e) \\ S(x)\ \texttt{catch cases} &\Rightarrow S(x) \\ F(e)\ \texttt{catch cases} &\Rightarrow \texttt{cases}(e) \\ S(x) + S(y) &\Rightarrow S(x+y) \\ F(e) + S(y) &\Rightarrow F(e) \\ S(x) + F(e) &\Rightarrow F(e) \end{aligned}$$

Solution 3: Try Type Combine advantages of `Option` and exceptions:

```
sealed abstract class Try[+A]
case class Success[+A](value: A) extends Try[A]
case class Failure(exc: Throwable) extends Try[Nothing]

object Try:
  def apply[A](e: => A): Try[A] =
    try Success(e)
    catch case exc => Failure(exc)
```

Usage:

```
def recip100(v: Int): Int =
  if v == 0 then throw new ReciprocalOfZero
  else 100 / v

def f(x: Int, y: Int): Try[Int] =
  Try(recip100(x) + recip100(y))

// Pattern matching on result
f(0, 5) match
  case Success(v) => println(s"Result: $v")
  case Failure(e) => println(s"Error: ${e.getMessage}")
```

Composing Try Values Use `flatMap` for cleaner composition:

```
sealed abstract class Try[+A]:
  def flatMap[B](onSuccess: A => Try[B]): Try[B] =
    this match
      case Failure(e) => Failure(e)
      case Success(v) => onSuccess(v)

def recip100(v: Int): Try[Int] =
  if v == 0 then Failure(new ReciprocalOfZero)
  else Success(100 / v)

def f(x: Int, y: Int): Try[Int] =
  recip100(x).flatMap: vx =>
    recip100(y).flatMap: vy =>
      Success(vx + vy)
```

Break Statements with Exceptions Scala provides controlled breaks using exceptions:

```
import scala.util.boundary, boundary.break

def firstIndex[T](xs: List[T], elem: T): Int =
  boundary:
    for (x, i) <- xs.zipWithIndex do
      if x == elem then break(i)
    -1

// break throws an exception caught by boundary
```

1.3.8 Control Flow Transformation

Program Counter Representation Any control flow can be represented using a program counter variable.

Original nested loops:

```
var i = 0
var j = 0
while i < 10 do
  j = 0
  while j < i do
    f(i, j)
    j += 1
  i += 1
```

Transformed with program counter:

```
var i = 0
var j = 0
var pc = 1
```

```
while pc != 6 do
  pc match
    case 1 =>
      if i < 10 then
        j = 0;
        pc = 2
      else pc = 6
    case 2 =>
      if j < i then pc = 3
      else pc = 5
    case 3 =>
      f(i, j); pc = 4
    case 4 =>
      j += 1; pc = 2
    case 5 =>
      i += 1; pc = 1
```

This technique enables implementing custom control flow (break, continue, goto) by manipulating pc.

General Recursion with Stack Transform recursive functions into iterative ones using explicit stack:

Original recursive evaluator:

```
def eval(expr: Expr): Int =
  expr match
    case Const(i) => i
    case Minus(e1, e2) =>
      val v1 = eval(e1)
      val v2 = eval(e2)
      v1 - v2
```

Stack implementation:

```
case class Stack[T](var content: List[T] = List()):
  def isEmpty: Boolean = content.isEmpty
  def push(v: T): Unit = content = v :: content
  def pop: T =
    val res = content.head
    content = content.tail
    res
```

Transformed with explicit stack:

```
def eval(expr: Expr): Int =
  var exprStack = Stack[Expr]()
  var resStack = Stack[Int]()
  var pcStack = Stack[Int]()
  var expr0 = expr
  var pc = 1

  while !(pcStack.isEmpty && pc == 4) do
    pc match
      case 1 =>
        expr0 match
          case Const(i) =>
            resStack.push(i)
            pc = 4
          case Minus(e1, e2) =>
            pcStack.push(2)
            exprStack.push(e2)
            expr0 = e1
            pc = 1
      case 2 =>
```

```
        pcStack.push(3)
        expr0 = exprStack.pop
        pc = 1
      case 3 =>
        val v2 = resStack.pop
        val v1 = resStack.pop
        resStack.push(v1 - v2)
        pc = 4
      case 4 =>
        if !pcStack.isEmpty then
          pc = pcStack.pop

  resStack.pop
```

This transformation:

- Eliminates recursion (avoids stack overflow)
- Makes control flow explicit
- Enables custom control strategies
- Used by compilers for code generation

Summary Functional interfaces with imperative implementations provide:

- Performance optimization through caching and mutation
- Maintained correctness via object invariants
- Abstraction of implementation details
- Flexibility in control flow representation

Key principle: Hide effects behind pure interfaces to gain efficiency without sacrificing reasoning capabilities.

1.3.9 Asynchronous Programming with Futures

The Problem Sequential code wastes time when operations can run independently:

```
def makeBreakfast(): (Coffee, Croissant) =
  val coffee = makeCoffee()      // takes 1 second
  val croissant = bakeCroissant() // takes 5 seconds
  (coffee, croissant)             // total: 6 seconds
```

We want to run independent tasks concurrently without blocking.

Evaluation Strategies

- **Strict (eager):** Evaluate when defined
- **Lazy:** Evaluate when needed
- **Lenient:** Can evaluate anytime after definition, must evaluate when needed

Lenient evaluation enables parallelism.

1.3.10 Part 1: Simple Futures

Basic Idea Separate task definition from result retrieval:

```
val f = SimpleFuture:
  /* some task */
// ... do other work ...
val result = f.await  // wait for result when needed
```

Example: Parallel Breakfast

```
def makeCoffee(): SimpleFuture[Coffee] =
  SimpleFuture:
    val beans = grindBeans().await
    brewCoffee(beans).await

def makeBreakfast(): SimpleFuture[(Coffee, Croissant)] =
  val coffeeFuture = makeCoffee()        // start coffee
  val croissantFuture = bakeCroissant()  // start croissant
  SimpleFuture:
    (coffeeFuture.await, croissantFuture.await)  // wait for both
```

Simple Implementation

```
class SimpleFuture[T](body: => T):
  private var status: Option[Try[T]] = None
  private val thread = new Thread:
    override def run(): Unit =
      status = Some(Try(body))
  start()

  def await: T =
    if status.isEmpty then thread.join()
    status.get.get
```

Problem: Thread Exhaustion Creating one thread per future doesn't scale:

- System limit: typically only a few thousand threads
- Blocking threads waste resources
- Web servers need many concurrent connections

1.3.11 Part 2: Completable Futures

Solution: Callbacks Use callbacks instead of blocking threads:

```
// Instead of blocking
def compute(): Result

// Use callback
def compute(callback: Result => Unit): Unit
```

Scheduler Tasks run from a scheduler, not dedicated threads:

```
object scheduler:
  private val tasks: ListBuffer[() => Unit] = ListBuffer()

  def schedule(task: => Unit) =
    tasks.append(() => task)

  def run(): Unit =
    while tasks.nonEmpty do
      val task = tasks.remove(0)
      task()
```

Callback-Based Code

```
def makeCoffee(callback: Coffee => Unit): Unit =
  grindBeans: beans =>
    brew(beans): coffee =>
      callback(coffee)

// Waiting for two callbacks in parallel
def makeBreakfast(serve: (Coffee, Croissant) => Unit) =
  var myCoffee: Option[Coffee] = None
  var myCroissant: Option[Croissant] = None

  makeCoffee: coffee =>
    myCroissant match
      case Some(croissant) => serve(coffee, croissant)
      case None => myCoffee = Some(coffee)

  bakeCroissant: croissant =>
    myCoffee match
      case Some(coffee) => serve(coffee, croissant)
      case None => myCroissant = Some(croissant)
```

Problems with Callbacks

- **Callback hell:** Code drifts rightward with nesting
- **Error handling:** Must propagate errors manually
- **Parallel composition:** Complex state management

Future Abstraction Transform callback style into cleaner API:

```
// Callback style
def program(a: A, callback: B => Unit): Unit

// Future style
def program(a: A): Future[B]
```

Where `Future[T]` represents an asynchronous computation returning `T`.

Future Trait

```
trait Future[+T]:
  def onComplete(callback: Try[T] => Unit): Unit

  def map[B](f: T => B): Future[B]
  def flatMap[B](f: T => Future[B]): Future[B]
  def zip[B](other: Future[B]): Future[(T, B)]
  def recover(f: Exception => T): Future[T]
```

Cleaner Code with Future

```
def makeCoffee: Future[Coffee] =
  for
    beans <- grindBeans
    coffee <- brew(beans)
  yield coffee

def makeBreakfast: Future[(Coffee, Croissant)] =
  makeCoffee.zip(bakeCroissant)

// Or with explicit parallelism
def makeBreakfast: Future[(Coffee, Croissant)] =
  val coffeeFuture = makeCoffee
```

```
  val croissantFuture = bakeCroissant
  for
    coffee <- coffeeFuture
    croissant <- croissantFuture
  yield (coffee, croissant)
```

Promise: Completing Futures

```
class Promise[T]:
  def complete(result: Try[T]): Unit
  val future: Future[T]

enum State[T]:
  case Pending(callbacks: List[T => Unit])
  case Complete(result: T)
```

A `Promise` completes a `Future` with a result.

Implementation Example: map

```
def map[U](f: T => U): Future[U] = new Future[U]:
  def onComplete(callback: Try[U] => Unit): Unit =
    Future.this.onComplete:
      case Success(x) => callback(Try(f(x)))
      case Failure(e) => callback(Failure(e))
```

Using Standard Library Futures

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def makeBreakfast: Future[(Coffee, Croissant)] =
  makeCoffee.zip(bakeCroissant)

// Blocking wait in main (avoid in production)
import scala.concurrent.{Await, duration}
Await.result(makeBreakfast, duration.Duration.Inf)
```

1.3.12 Part 3: Direct Style with Continuations

Modern Trend Runtimes now support lightweight concurrency primitives:

- Go: Goroutines
- Java 21+: Virtual Threads
- Kotlin: Coroutines
- Scala Native: Continuations

This enables returning to direct style without blocking threads.

Boundary and Break

```
import scala.util.boundary, boundary.break

def firstIndex[T](xs: List[T], elem: T): Int =
  boundary:
    for (x, i) <- xs.zipWithIndex do
      if x == elem then break(i)
    -1
```

`boundary` establishes a scope, `break` returns from it.

Suspensions Delimited continuations allow suspending and resuming execution:

```
class Suspension[-T, +R]:
  def resume(arg: T): R

def suspend[T, R](body: Suspension[T, R] => R)(using Label[R]): T
```

Direct-Style Futures Combine simplicity of direct style with efficiency of async:

```
val sum = Future:
  val f1 = Future(c1.read)
  val f2 = Future(c2.read)
  f1.await + f2.await  // await without blocking threads
```

Structured concurrency: Local futures complete before parent completes.

Comparison of Approaches

Aspect	Simple	Completable	Direct-Style
Syntax	Natural	Verbose	Natural
Scalability	Poor	Good	Good
Threads	One per future	Shared pool	Virtual/fibers
Learning curve	Easy	Medium	Easy
Composability	Good	Medium	Excellent

Summary Three approaches to asynchronous programming:

1. **Simple futures:** Easy but doesn't scale (thread per task)
2. **Completable futures:** Scalable but complex (callbacks, monads)
3. **Direct-style futures:** Best of both (requires runtime support)

Modern trend: Move toward direct style with lightweight concurrency primitives.

Key insight: Separate when computations start from when results are needed to enable parallelism.

2 Software engineering

2.1 Version control

Plain backups Tools such as Google Drive, Syncthing, Dropbox, or Nextcloud are useful for file synchronization, but they are not designed for software development.

- No meaningful change sets
- Too large or too small granularity
- Limited history browsing capabilities
- Limited support for asynchronous collaboration

Version Control Systems (VCSs) Also called Source Code Management systems (SCMs).

- Tracking the evolution of text or code (Labeled incremental snapshot).
- Versioning, distributed development.

Software development with a VCS typically involves the following steps:

1. Write code
2. Debug and test code
3. Review code changes
4. Write description of changes
5. Save snapshot

2.1.1 Git

Git is one of the most popular distributed version control systems.

.git directory Internal Git database. Created automatically when running `git init`. Never edit or commit it manually.

.gitignore Defines which files and folders Git must not track (e.g., `target/`, `.metals/`). Exclusions can use wildcards and are relative to the location of the file. Multiple `.gitignore` files are possible in different directories.

Configuration and help

```
# Show help for Git commands
git help
git help <command>        # detailed help for a specific command

# Configure user information
git config --global user.name "Name"
git config --global user.email "email@example.com"
git config --global core.editor "code"  # set default editor

# Better merge conflict resolution
git config --global merge.conflictstyle diff3
# Shows: your changes | common ancestor | incoming changes

# View configuration
git config --list         # show all configuration
git config user.name      # show specific configuration
```

Repository initialization and cloning

```
# Initialize a new repository
git init                  # create .git directory in current folder

# Clone an existing repository
git clone <url>           # clone from remote URL
git clone <url> <dir>     # clone into specific directory
git clone --recurse-submodules <url>  # clone with submodules
git clone <bundle-file>   # clone from bundle file
git clone <repos-path>    # clone from local path
```

Basic workflow

```
# Check repository status
git status                # show working tree status
git status -s             # short format

# Show changes
git show                  # show latest commit
git show <commit>         # show specific commit
git show HEAD~1           # show parent of HEAD
```

```
# Compare changes
git diff                    # unstaged changes
git diff --staged           # staged changes (ready to commit)
git diff HEAD               # all changes (staged + unstaged)
git diff <commit>           # changes since specific commit
git diff <branch>           # changes compared to another branch
git diff <commit1> <commit2>  # changes between two commits
git diff main..feature      # changes between branches
git diff --stat             # show summary statistics
git diff --name-only        # show only filenames
git diff --color-words      # word-level diff
git diff --word-diff=color # word-level diff with colors
git diff --ignore-all-space # ignore whitespace changes

# Stage changes
git add <file>              # stage specific file
git add .                   # stage all changes in current directory
git add -A                  # stage all changes in repository
git add -p                  # interactively stage chunks
git add -i                  # interactive staging

# Commit changes
git commit -m "message"     # commit with message
git commit --amend          # modify last commit
git commit --amend --no-edit  # add changes to last commit without editing message
git commit -a -m "msg"      # stage and commit tracked files
```

History and inspection

```
# View commit history
git log                     # show commit history
git log --oneline           # compact one-line format
git log --graph             # show branch graph
git log --all --graph       # show all branches
git log -n 5                # show last 5 commits
git log --since="2 weeks" # commits from last 2 weeks
git log --author="Name"     # commits by specific author
git log -- main.scala       # history of specific file
git log -L 1,50:main.scala  # history of lines 1-50 in file
git log --follow <file>     # follow file through renames
git log --graph --author='Author' --patch <commit1>..<commit2>

# Summarize commits
git shortlog                # group commits by author
git shortlog -sn            # count commits per author, sorted

# Show who modified each line
git blame <file>            # show author and commit for each line
git blame -L 10,20 <file> # blame specific line range

# Advanced history comparison
git range-diff main@{1} main@{0} feature
# Shows how commits evolved between rebases
git range-diff old-feature new-feature main
# Compare how feature branch changed after rebase
git range-diff upstream/main @{upstream} HEAD
# Useful for reviewing rebased branches
```

Branching and navigation

```
# List branches
git branch                  # list local branches
git branch -a               # list all branches (local + remote)
```

```
git branch -v             # show last commit on each branch
git branch -d <branch>    # delete branch (safe)
git branch -D <branch>    # force delete branch

# Create and switch branches
git branch <name>         # create new branch
git checkout <branch>     # switch to branch
git checkout -b <name>    # create and switch to new branch
git switch <branch>       # modern way to switch branches
git switch -c <name>      # create and switch (modern syntax)

# Navigate history
git checkout <commit>     # checkout specific commit (detached HEAD)
git checkout HEAD~1       # checkout parent commit
git checkout HEAD~3       # checkout 3 commits back
git checkout main         # return to main branch

# Merge branches
git merge <branch>           # merge branch into current branch
git merge --no-ff <branch>   # force merge commit
git merge --ff-only <branch> # fast-forward only (no merge commit)
git merge --squash <branch>  # squash all commits into one
git merge --abort            # abort merge in progress
```

Remote repositories

```
# Manage remotes
git remote                # list remotes
git remote -v             # show remote URLs
git remote add <name> <url>  # add new remote
git remote remove <name>  # remove remote
git remote rename <old> <new> # rename remote

# Fetch and pull
git fetch                 # download objects from remote
git fetch <remote>        # fetch from specific remote
git fetch --all           # fetch from all remotes
git pull                  # fetch and merge
git pull --rebase         # fetch and rebase

# Push changes
git push                  # push to default remote
git push <remote> <branch>  # push to specific remote/branch
git push -u origin main   # push and set upstream
git push --force          # force push (dangerous!)
git push --force-with-lease  # safer force push
git push --delete origin <branch> # delete remote branch

# Working with forks
# 1. Fork on GitHub/GitLab (use web interface)
# 2. Clone your fork
git clone <your-fork-url>
# 3. Add upstream remote (original repository)
git remote add upstream <original-repo-url>
# 4. Keep fork updated
git fetch upstream
git merge upstream/main
# 5. Push to your fork
git push origin main

# Offline workflows
git bundle create repo.bundle --all           # bundle entire repository
git bundle create repo.bundle main            # bundle specific branch
git bundle create repo.bundle main..feature   # bundle commits range
git bundle verify repo.bundle                 # verify bundle integrity
```

```
git clone repo.bundle new-repo                # clone from bundle
git pull repo.bundle main                     # pull updates from bundle
```

Undoing changes

```
# Restore files
git restore <file>        # discard changes in working directory
git restore --staged <file>  # unstage file
git restore --source=HEAD~1 <file>  # restore from specific commit

# Reset commits (moves HEAD and branch pointer)
git reset <commit>        # move HEAD, keep changes in working directory (--mixed)
git reset --soft <commit> # move HEAD, keep changes staged
git reset --hard <commit> # move HEAD, discard all changes (dangerous!)
git reset HEAD~1          # undo last commit, keep changes
git reset --hard HEAD~3   # go back 3 commits, lose all changes

# Revert commits (creates new commit)
git revert <commit>       # create new commit that undoes changes
git revert HEAD           # revert last commit
git revert <commit1>..<commit2> # revert range of commits
git revert --no-commit <commit> # revert without auto-commit

# Recover lost commits with reflog
git reflog                # show all HEAD movements
git reflog show <branch>  # show branch history
git reset --hard HEAD@{2} # go back to previous state
git checkout HEAD@{5}     # checkout old state
```

Stashing

```
# Temporarily save changes
git stash                 # stash current changes
git stash save "message"  # stash with message
git stash -u              # include untracked files
git stash --all           # include ignored files too

# Manage stashes
git stash list            # list all stashes
git stash show            # show stash content
git stash show -p         # show stash diff
git stash pop             # apply and remove latest stash
git stash apply           # apply stash without removing
git stash apply stash@{2} # apply specific stash
git stash drop            # remove latest stash
git stash drop stash@{1}  # remove specific stash
git stash clear           # remove all stashes
```

Rebasing Rebase rewrites history by moving commits to a new base.

```
# Basic rebase
git rebase <branch>       # rebase current branch onto another
git rebase main           # rebase current branch onto main
git rebase --continue     # continue after resolving conflicts
git rebase --skip         # skip current commit
git rebase --abort        # abort rebase

# Interactive rebase
git rebase -i HEAD~3      # interactive rebase last 3 commits
git rebase -i <commit>    # rebase from specific commit

# Interactive rebase commands:
# pick   - use commit as-is
# reword - use commit, but edit message
```

```
# edit   - use commit, but stop for amending
# squash - combine with previous commit, edit message
# fixup  - combine with previous commit, discard message
# drop   - remove commit

# Rebase onto different base
git rebase --onto main feature-old feature-new
```

Rebase vs Merge:

Aspect	Merge	Rebase
History	Preserves all history	Creates linear history
Commits	Creates merge commit	Rewrites commits
Conflicts	Resolve once	May resolve multiple times
Use when	Public branches	Local cleanup
Safety	Safe (non-destructive)	Dangerous (rewrites history)

Cherry-picking Apply specific commits to current branch.

```
# Cherry-pick commits
git cherry-pick <commit>  # apply specific commit to current branch
git cherry-pick <commit1> <commit2> # apply multiple commits
git cherry-pick <commit1>..<commit2> # apply range of commits
git cherry-pick --continue # continue after resolving conflicts
git cherry-pick --abort    # abort cherry-pick
git cherry-pick --no-commit <commit> # apply without committing
```

Patches Create and apply patches for sharing changes without direct repository access.

```
# Create patches
git format-patch HEAD~3   # create patches for last 3 commits
git format-patch <branch> # create patches for commits not in branch
git format-patch -1       # create patch for last commit
git format-patch -3       # create patches for last 3 commits
git format-patch main..feature                # create patches for commit range
git format-patch -o patches/ main..feature    # output patches to directory

# Apply patches
git apply <patch>                             # apply patch file
git apply --check patch.patch                 # check if patch can be applied
git apply --stat patch.patch                  # show patch statistics
git am <patch>                                # apply patch with commit info
git am < 0001-commit-message.patch            # apply patch with commit info
git am --continue                             # continue after resolving conflicts
git am --abort                                # abort patch application
```

Tags Tags mark specific points in history (releases, milestones).

```
# List and show tags
git tag                  # list tags
git tag -l "v1.*"        # list tags matching pattern
git show <tag>           # show tag details

# Create tags
git tag <name>           # create lightweight tag
git tag -a <name> -m "msg"  # create annotated tag
git tag -a <name> <commit> # tag specific commit

# Push and delete tags
git push origin <tag>     # push tag to remote
git push --tags           # push all tags
git push --delete origin <tag> # delete remote tag
git tag -d <tag>          # delete local tag
```

Cleaning Remove untracked files and directories.

```
# Clean working directory
git clean -n              # dry run: show what would be removed
git clean -f              # remove untracked files
git clean -fd             # remove untracked files and directories
git clean -fdx            # also remove ignored files
git clean -fdi            # interactive mode
```

Advanced conflict resolution Handle merge conflicts effectively.

```
# Configure diff3 style (recommended)
git config --global merge.conflictstyle diff3
# Shows three sections in conflicts:
# <<<<<<< HEAD (your changes)
# ||||||| common ancestor
# ======= incoming changes
# >>>>>>> branch-name

# During conflicts
git status                # see conflicted files
git diff                  # show conflicts
git mergetool             # launch visual merge tool
git checkout --ours <file>   # keep our version
git checkout --theirs <file> # keep their version
git add <file>            # mark as resolved
git merge --abort         # abort merge
git rebase --abort        # abort rebase

# After resolving
git add <resolved-files>
git commit                # for merge
git rebase --continue     # for rebase
git cherry-pick --continue # for cherry-pick
```

Detached HEAD state Working with commits not on any branch.

```
# Enter detached HEAD state
git checkout <commit-hash>

# Create branch from detached HEAD
git switch -c <new-branch-name>
git checkout -b <new-branch-name>

# Return to normal state
git switch <branch-name>
git checkout <branch-name>

# Warning: commits made in detached HEAD will be lost
# unless you create a branch before switching away
```

Submodules Submodules allow you to include external repositories as subdirectories within your project.

```
# Add and initialize submodules
git submodule add <url> <path>   # add submodule at path
git submodule add https://github.com/user/lib.git libs/mylib
git submodule update --init       # initialize submodules
git submodule update --init --recursive  # including nested submodules

# Update submodules
git submodule update --remote     # update submodules to latest
git submodule update --remote --merge
git submodule foreach 'git pull origin main'  # execute command in all
```

```
# Remove submodule
git submodule deinit <path>
git rm <path>
rm -rf .git/modules/<path>
```

Subtree Subtree is an alternative to submodules, merging external repositories directly into your tree.

```
# Add subtree (one-time setup)
git subtree add --prefix=libs/mylib <url> main --squash

# Pull updates from subtree
git subtree pull --prefix=libs/mylib <url> main --squash

# Push changes back to subtree repository
git subtree push --prefix=libs/mylib <url> main

# Split out subtree into separate repository
git subtree split --prefix=libs/mylib -b mylib-branch
```

Aspect	Submodules	Subtree
Complexity	More complex	Simpler for users
Repository size	Smaller	Larger
History	Separate	Merged
Updates	Explicit commands	Standard pull
Best for	Clear separation	Tight integration

Email workflow Used by projects like the Linux kernel that use mailing list-based development.

```
# Configure email
git config --global sendemail.smtpserver smtp.gmail.com
git config --global sendemail.smtpserverport 587
git config --global sendemail.smtpencryption tls
git config --global sendemail.smtpuser your@email.com

# Send patches via email
git send-email --to=maintainer@project.org 0001-*.patch

# Generate patches with cover letter then send
git format-patch -3 --cover-letter
git send-email --to=list@project.org *.patch
```

Useful aliases Add these to your `.gitconfig` for shortcuts:

```
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.unstage 'reset HEAD --'
git config --global alias.last 'log -1 HEAD'
git config --global alias.lg "log --graph --oneline --all"
git config --global alias.undo "reset --soft HEAD~1"
```

Best Practices

- Commit often with clear, descriptive messages
- Use branches for features and experiments
- Never rebase public/shared branches

- Review changes before committing (`git diff`)
- Pull before push to avoid conflicts
- Use `.gitignore` to exclude build artifacts
- Configure `diff3` conflict style for better conflict resolution
- Use `git reflog` to recover from mistakes

2.2 Debugging

Process

Triage

1. Check that there is a problem (Confirm that there is an issue)
    - State what the issue is. (May be refined later) => "Coursier exits silently without installing Scala"
    - Compare to the spec, the documentation the requirements. => "find -type <f> should not return directories"
    - Is it obvious why the bad case is different from good case.
2. Reproduce the issue
    - Does it happen every time
    - Does it happen for every input => "Only when clicking repeatedly"
    - Does it depend on system config => "Only with a specific version"
    - Are there any diagnostics => "Error message, logs, etc. . . "
    - Did it work at one point => "Previous versions, commit History"
3. Decide whether it's a problem (Not alls bugs are worth fixing)
    - Do / Need to Fix ?
    - Is it worth fixing ? => "Workaround may be sufficient"
    - Does it need to be fixed now ?
    - Do I know where to complain ?
4. Write it Up
    - Check previous reports => "Bugs DB, Questions etc. . . "
    - Where to report => "Email, bug tracker, contact form"
    - Check reporting guideline => "Is there a security policy"
    - Write clearly and completely => "Helpful title, clear problem, expected behavior, reproduction steps, relevant info present"

Diagnose and fix

1. Learn about the system
    - Consult the docs/manual

- Search for relevant resource
- Know what you do not know => “Does 1 until n include n ?”
- Know the relevant tools => “JTAG, perf, strace, objdumps, a multimeter”
- Skim to the code => “Find entry point, seemingly relevant functions”

2. Observe the issue
    - Exercise different angles => “Vary the inputs, look upstream and downstream (consequence)”
    - Read error messages
    - Add logging / tracing
    - Use the right tools
    - Use the VCS history
    - Read the code
3. Simplify, minimize, and isolate
    - Simplify the inputs
    - Simplify the system
    - Slow things down
    - Determinism the failure => “One process, set random seed”
    - Automate the failure => “Unit test”
4. Guess and verify
    - Formulate a hypothesis => “I forgot to clamp the speed”
    - Design and experiment => => “Transform or observe to test the hypothesis (Logging, breakpoints, assertions, prints)”
    - Narrow down issue => “Divide into smaller bugs, look for the root cause”
5. Fix and confirm the fix
    - Decide whether to fix the problem => “Easy to fix ? Workaround preferable ?”
    - Apply the changes
    - Revert other changes => “Confirm fix on clean system”
    - Confirm the fix => “All tests pass, no new bug”
6. Prevent regressions
    - Document the resolution => “Write detailed commit message, and update the documentation”
    - Look for similar instances
    - Add missing tests’

Techniques

- Keep notes
- Change one thing at a time
- Apply the scientific method
- Instrument
- Divide and conquer
- Ask for help

Pitfalls

- Random mutation
- Staring aimlessly
- Wasting time
- Assuming a bug went Always
- Fixing effects, not causes
- Losing data

2.3 Testing

2.3.1 Testing Overview

Test Types

Acceptance testing Customer validates the system meets requirements

System testing Developer validates system with comprehensive checklists

Integration testing Tests interaction between multiple components

Unit tests Tests individual components in isolation

Monitors Runtime verification using pre/post conditions

Traditional Testing Approach Automated test consists of three elements:

1. **System under test (SUT):** The code being tested
2. **Input:** Test data provided to the system
3. **Expectation:** Specification of correct behavior

Types of Expectations

- **Model-based:** Compare against reference implementation

```
List(1,2,1).distinctWithHashMap == List(1,2)
List(1,3,2).quickSort == List(1,2,3)
```

- **Axiomatic:** Verify properties hold

```
// noDuplicates property
List(1,2,1).distinctWithHashMap
// isSorted property
List(1,3,2).sort
```

2.3.2 Limitations of Unit and Integration Tests

Traditional unit and integration tests have several drawbacks:

- **Tedious and time-consuming:** Writing individual tests for each case requires significant effort
- **Basic tests crowd out interesting tests:** Time spent on trivial tests reduces coverage of edge cases
- **Incomplete coverage:** Developers must anticipate the right inputs, which is difficult
- **Regression vs comprehensive testing:** Regression tests (verifying known issues don't recur) are easy, but comprehensive tests are hard

Key insight: Each unit/integration test validates behavior for exactly one input.

2.3.3 Automated Testing with Monitors

Definition Monitors are runtime checks that validate specifications during normal program execution. They transform one monitor specification into infinitely many tests over the application's lifetime.

Specifications for Monitors

- **Model-based specification:** Compare output against reference implementation

```
ls.distinctWithHashMap.ensuring(r => r == ls.distinct)
ls.quickSort.ensuring(r => r == ls.sorted)
```

- **Axiomatic specification:** Verify properties of output

```
ls.distinctWithHashMap.ensuring(r => noDuplicates(r))
ls.quickSort.ensuring(r => isSorted(r))
```

Key Advantage Monitors enable testing individual components using integration runs and real executions:

- One unit test = one input/output pair
- One monitor = infinitely many tests throughout application lifetime

Development Workflow Comparison Traditional approach with unit/integration tests:

1. Write code
2. Think hard about interesting inputs
3. Write unit tests and integration tests
4. Run tests and debug

Approach with monitors:

1. Write code
2. Think hard about interesting properties
3. Write monitors
4. Run application and debug

Example: Detecting Errors in Production Consider a function to normalize strings:

```
/** Removes diacritics and non-alphabetic characters from `s`. */
def normalizeString(str: String): String =
  Normalizer.normalize(str, Normalizer.Form.NFD)
    .replaceAll("\\p{InCombiningDiacriticalMarks}+", "")
    .replaceAll("[^a-zA-Z]+", "")
    .toLowerCase
ensuring (_.forall(c => 'a' <= c && c <= 'z'))
```

Problems detected by monitoring:

1. **Not pure:** Uses implicit default locale

   ```
   import java.util.Locale
   Locale.setDefault(Locale.forLanguageTag("tr"))
   "I".toLowerCase  // Returns: i (Turkish dotless i)
   "i".toUpperCase  // Returns: I (Turkish capital i with dot)
   ```

2. **Not properly tested:** Would require testing all locales with many strings

Fixed version:

```
def normalizeString(str: String)(using locale: Locale): String =
  Normalizer.normalize(str, Normalizer.Form.NFD)
    .replaceAll("\\p{InCombiningDiacriticalMarks}+", "")
    .replaceAll("[^a-zA-Z]+", "")
    .toLowerCase(locale)
ensuring (_.forall(c => 'a' <= c && c <= 'z'))
```

2.3.4 Property-Based Testing

Core Idea Generate synthetic inputs automatically to validate specifications, rather than manually writing individual test cases.

Basic Usage with ScalaCheck ScalaCheck provides the `forAll` construct to test properties:

```
forAll((x: Int) => x + 1 - 1 == x).check()

forAll { (l: List[Int]) =>
  l.reverse == l.foldLeft(Nil)((acc, x) => x :: acc)
}.check()

forAll { (l: List[Int]) =>
  l.reverse == l.foldRight(Nil)((x, acc) => x :: acc)
}.check()
```

Handling Preconditions Use the **==>** operator to express preconditions:

```
// Wrong: will fail for empty lists
forAll { (l: List[Int]) =>
  l.head :: l.tail == l
}.check()

// Correct: add precondition
forAll { (l: List[Int]) =>
  (l != Nil) ==> (l.head :: l.tail == l)
}.check()

// Example with overflow protection
forAll { (x: Int) =>
  (x != Int.MaxValue) ==> (x + 1 > x)
}.check()
```

Testing State Machines State machines can be tested with property-based testing because they are pure:

- **Input:** Sequence of events
- **Specifications:**
  - Model-based: Function of all events
  - Axiomatic: Property of the resulting state

ScalaCheck provides custom support for testing state machines.

2.3.5 Beyond Property-Based Testing

When specifications are difficult to write or unavailable, alternative testing approaches can be used.

Differential Testing Compare two implementations (systems under test) against each other:

```
ls.quickSort == ls.mergeSort
```

Advantage: Similar to model-based testing, but neither implementation needs to be proven correct.

Limitation: Both implementations might have the same bug.

Mutational Testing Change inputs in ways that should not affect output:

```
eval(e) == eval(Plus(e, 0)) == eval(Times(e, 1))
```

Advantage: Tests semantic equivalence properties.

Use case: Verifying optimization correctness.

Crash Testing (Fuzzing) Use "does not crash" as the specification:

```
try { eval(e) } catch { case _ => "Test failed!" }
```

Advantage: Finds unexpected failures without needing detailed specifications.

Common use: Security testing and robustness verification.

2.3.6 Beyond Generators: Fuzzing Techniques

When creating custom input generators is difficult, fuzzing techniques can automatically explore the input space.

Black-Box Fuzzing Explore bit patterns without understanding program structure:

- Programs work with bytes, so no custom generators needed
- Generate random or mutated byte sequences
- Feed directly to program entry points

Advantage: Simple to implement, no program knowledge required.

Limitation: Inefficient for programs with complex input validation.

Grey-Box Fuzzing (Coverage-Guided) Use instrumentation to maximize code coverage:

1. Record program execution paths
2. Identify inputs that reach new code
3. Prioritize mutations of interesting inputs
4. Iterate to maximize coverage

Advantage: Much more effective than random fuzzing.

Example tools: AFL (American Fuzzy Lop), LibFuzzer.

White-Box Fuzzing (Concolic Execution) Use symbolic execution and constraint solvers:

1. Execute program symbolically to track constraints
2. Collect path conditions (branches taken)
3. Use SMT solver to generate inputs for unexplored branches
4. Systematically explore all paths

Advantage: Can reverse-engineer inputs to reach specific code paths.

Limitation: Does not scale well to large programs due to path explosion.

Example tools: KLEE, SAGE, Mayhem.

2.3.7 Summary: Testing Approaches

Approach	Specification Needed	Input Generation
Unit/Integration Tests	Yes	Manual
Monitors	Yes	From real usage
Property-Based Testing	Yes	Automatic generators
Differential Testing	No (two implementations)	Automatic generators
Mutational Testing	Partial (equivalences)	Transformation rules
Crash Testing	No (just detect crashes)	Automatic generators
Black-box Fuzzing	No	Random bytes
Grey-box Fuzzing	No	Coverage-guided
White-box Fuzzing	No	Constraint solving

Key Takeaways

- **Monitors** extend testing from single inputs to all program executions
- **Property-based testing** automates input generation while maintaining strong specifications
- **Differential and mutational testing** reduce specification burden
- **Fuzzing** finds bugs without specifications, using various sophistication levels
- **Choose the right approach:** Balance specification effort, input generation complexity, and coverage goals

2.4 Functional Interfaces with Imperative Implementations

The strategy is to implement a pure functional interface using internal mutation for efficiency, while hiding implementation details from external code.

Goal Replace a pure but inefficient function `f` with an optimized version `f'` such that:

1. `f'` is more efficient than `f`
2. `f'` uses mutation internally for performance
3. `f'(x)` returns the same value as `f(x)` for all `x`

If the implementation is correct, replacing `f` with `f'` preserves program behavior while improving performance.

2.4.1 Caching Patterns

Lazy Values Review Scala provides lazy evaluation with `lazy val`:

```
// Function: evaluated every time
val x = () => {println("Evaluating x"); 42}
x()  // Evaluating x
x()  // Evaluating x (again)

// Lazy val: evaluated once
lazy val y = {println("Evaluating y"); 42}
y    // Evaluating y
y    // (no output, cached result)
```

LazyCell Class A `LazyCell` encapsulates lazy evaluation:

```
class LazyCell[+A](init: => A):
  lazy val get = init

val lc = LazyCell({println("Computing"); 42})
lc.get  // Computing
        // 42
lc.get  // 42 (no recomputation)
```

LazyCell with Mutation Implementation using internal mutation for efficiency:

```
class LazyCell[+A](val init: () => A):
  private var cached: Option[A] = None

  def get: A =
    cached match
      case Some(a) => a
      case None =>
        cached = Some(init())
        cached.get
```

This implements the pure interface:

```
class LazyCell[+A](val init: () => A):
  def get: A = init()
```

Correctness: Object Invariant The `LazyCell` maintains an invariant ensuring correctness:

```
class LazyCell[+A](val init: () => A):
  private var cached: Option[A] = None

  def valid: Boolean =
    cached == None || cached == Some(init())  // invariant

  def get: A =
    require(valid)
    cached match
      case Some(a) => a
```

```
      case None =>
        cached = Some(init())
        cached.get
  .ensuring(res => valid && res == init())
```

Invariant: `cached == None || cached == Some(init())`

Proof sketch (induction on execution steps):

- Initially: `cached == None` (constructor)
- If a step doesn't modify `cached`, invariant holds
- If a step modifies `cached`, it must be in `get` (private field)
- In `get`: `cached` becomes `Some(init())`, preserving invariant

Cached Function (Memoization) Generalize `LazyCell` to cache function results:

```
case class CachedFunction[-A, +B](val f: A => B):
  private var cache: Map[A, B] = Map()

  def apply(a: A): B =
    cache.get(a) match
      case Some(b) =>
        println(s"Cache hit: $a -> $b")
        b
      case None =>
        val b = f(a)
        cache = cache.updated(a, b)
        b

val csin = CachedFunction(math.sin)
csin(0.4)  // 0.3894183423086505
csin(0.4)  // Cache hit: 0.4 -> 0.3894183423086505
```

Invariant

```
def valid: Boolean =
  cache.keys.forall(a => cache.get(a) == Some(f(a)))
```

All cached values must equal `f(a)` for their key `a`.

Aliasing Risk Exposing mutable state breaks correctness:

```
case class CachedFunction[-A, +B](val f: A => B):
  private var cache: Map[A, B] = Map()
  def getCache: Map[A, B] = cache  // BREAKS CORRECTNESS!
```

External code could modify the cache, violating the invariant.

2.4.2 Memoization for Recursive Functions

Fibonacci Example Naive recursive Fibonacci has exponential complexity:

```
def fib(n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else fib(n - 1) + fib(n - 2)

// Complexity: O(fib(n)) >= O(2^(n/2))
```

Applying `CachedFunction` to `fib` doesn't help:

```
val cf = CachedFunction(fib)
cf(44)  // Still slow! Only caches final result, not intermediate calls
```

Memoizing Recursive Calls Cache intermediate results by intercepting recursive calls:

```
var cache: Map[Int, Int] = Map()

def fib(n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else memo_fib(n - 1) + memo_fib(n - 2)

def memo_fib(a: Int): Int =
  cache.get(a) match
    case Some(b) => b
    case None =>
      val b = fib(a)
      cache = cache.updated(a, b)
      b

// Now O(n) time complexity
```

Abstracting Recursion Separate the recursive structure from the memoization logic:

```
// Recursor: parameterizes recursive calls
def fibR(rec: Int => Int, n: Int): Int =
  if n == 0 then 0
  else if n == 1 then 1
  else rec(n - 1) + rec(n - 2)

// Generic memoization
def memo(H: (Int => Int, Int) => Int): Int => Int =
  val cache: scala.collection.mutable.Map[Int, Int] =
    scala.collection.mutable.Map()

  def rec(a: Int): Int =
    cache.get(a) match
      case Some(b) => b
      case None =>
        val b = H(rec, a)
        cache(a) = b
        b
  rec

// Combine them
def fib(x: Int) = memo(fibR)(x)
```

Generic Memoization Generalize to any types:

```
def memo[A, B](H: (A => B, A) => B): A => B =
  val cache: scala.collection.mutable.Map[A, B] =
    scala.collection.mutable.Map()

  def rec(a: A): B =
    cache.get(a) match
      case Some(b) => b
      case None =>
        val b = H(rec, a)
        cache(a) = b
        b
  rec
```

`memo` takes a recursor `H` and creates a memoized function `f` such that `H(f, x) == f(x)` for all `x`.

2.4.3 Dynamic Programming

Concept Dynamic programming solves problems bottom-up, computing smaller subproblems first:

- Memoization: top-down (compute on demand, cache results)
- Dynamic programming: bottom-up (compute in order, store in array/table)

Advantages:

- No cache lookup overhead
- Predictable memory usage
- Often uses arrays instead of maps

Example: Floyd-Warshall Algorithm Find shortest paths between all pairs of vertices in a weighted graph.

Problem: Given directed graph with distances `d(from, to)`, find shortest path between every pair of nodes.

Recursive definition:

```
// path(from, to, k): shortest path using only nodes 0..k-1 as intermediates
def path(from: Int, to: Int, k: Int): Int =
  if k == 0 then d(from, to)
  else min(
    path(from, to, k - 1),              // don't use node k
    path(from, k, k - 1) + path(k, to, k - 1)  // go through k
  )
```

Complexity: Exponential (3 recursive calls).

Memoization approach: Store results in $O(N^3)$ table.

Dynamic programming approach: Use only $O(N^2)$ space by computing layer by layer:

```
def floydWarshall(d: Array[Array[Int]]): Array[Array[Int]] =
  val N = d.length
  var p = d.map(_.clone())  // current distances

  var k = 0
  while k < N do
    p = updateDistances(p, k)
    k += 1
  p

def updateDistances(p: Array[Array[Int]], k: Int): Array[Array[Int]] =
  val N = p.length
  val newP = p.map(_.clone())

  var from = 0
  while from < N do
    var to = 0
    while to < N do
      newP(from)(to) = min(
        p(from)(to),
        p(from)(k) + p(k)(to)
      )
      to += 1
    from += 1
  newP
```

Time: $O(N^3)$, **Space:** $O(N^2)$.

2.4.4 Exceptions

Problem with Partial Functions Some functions are undefined for certain inputs:

```
def recip100(v: Int): Int =
  100 / v

def f(x: Int, y: Int): Int =
  recip100(x) + recip100(y)

// f(0, 5) crashes with ArithmeticException
```

Solution 1: Option Type

```
def recip100(v: Int): Option[Int] =
  if v == 0 then None
  else Some(100 / v)

def f(x: Int, y: Int): Option[Int] =
  recip100(x) match
    case None => None
    case Some(vx) =>
      recip100(y) match
        case None => None
        case Some(vy) => Some(vx + vy)
```

Drawback: Verbose, nested pattern matching.

Solution 2: Exceptions

```
class ReciprocalOfZero extends Exception

def recip100(v: Int): Int =
  if v == 0 then throw new ReciprocalOfZero
  else 100 / v

def f(x: Int, y: Int): Int =
  recip100(x) + recip100(y)

// Caller handles exception:
try
  f(0, 5)
catch
  case _: ReciprocalOfZero => println("Division by zero")
```

Advantage: Concise code, error handling separate.

Exception Evaluation Rules Expressions evaluate to success `S(value)` or failure `F(exception)`:

```
throw e                    ==> F(e)
S(x) catch cases           ==> S(x)
F(e) catch cases           ==> cases(e)

S(x) + S(y)                ==> S(x + y)
F(e) + S(y)                ==> F(e)
S(x) + F(e)                ==> F(e)
```

Solution 3: Try Type Combine advantages of `Option` and exceptions:

```
sealed abstract class Try[+A]
case class Success[+A](value: A) extends Try[A]
case class Failure(exc: Throwable) extends Try[Nothing]
```

```
object Try:
  def apply[A](e: => A): Try[A] =
    try Success(e)
    catch case exc => Failure(exc)
```

Usage:

```
def recip100(v: Int): Int =
  if v == 0 then throw new ReciprocalOfZero
  else 100 / v

def f(x: Int, y: Int): Try[Int] =
  Try(recip100(x) + recip100(y))

// Pattern matching on result
f(0, 5) match
  case Success(v) => println(s"Result: $v")
  case Failure(e) => println(s"Error: ${e.getMessage}")
```

Composing Try Values Use `flatMap` for cleaner composition:

```
sealed abstract class Try[+A]:
  def flatMap[B](onSuccess: A => Try[B]): Try[B] =
    this match
      case Failure(e) => Failure(e)
      case Success(v) => onSuccess(v)

def recip100(v: Int): Try[Int] =
  if v == 0 then Failure(new ReciprocalOfZero)
  else Success(100 / v)

def f(x: Int, y: Int): Try[Int] =
  recip100(x).flatMap: vx =>
    recip100(y).flatMap: vy =>
      Success(vx + vy)
```

Break Statements with Exceptions Scala provides controlled breaks using exceptions:

```
import scala.util.boundary, boundary.break

def firstIndex[T](xs: List[T], elem: T): Int =
  boundary:
    for (x, i) <- xs.zipWithIndex do
      if x == elem then break(i)
    -1

// break throws an exception caught by boundary
```

2.4.5 Control Flow Transformation

Program Counter Representation Any control flow can be represented using a program counter:

Original nested loops:

```
var i = 0
var j = 0
while i < 10 do
  j = 0
  while j < i do
    f(i, j)
    j += 1
  i += 1
```

Transformed with program counter:

```
var i = 0
var j = 0
var pc = 1

while pc != 6 do
  pc match
    case 1 =>
      if i < 10 then {j = 0; pc = 2}
      else pc = 6
    case 2 =>
      if j < i then pc = 3
      else pc = 5
    case 3 =>
      f(i, j); pc = 4
    case 4 =>
      j += 1; pc = 2
    case 5 =>
      i += 1; pc = 1
```

This technique enables implementing custom control flow (break, continue, goto) by manipulating pc.

General Recursion with Stack Transform recursive functions into iterative ones using explicit stack:

Original recursive evaluator:

```
def eval(expr: Expr): Int =
  expr match
    case Const(i) => i
    case Minus(e1, e2) =>
      val v1 = eval(e1)
      val v2 = eval(e2)
      v1 - v2
```

Transformed with explicit stack:

```
case class Stack[T](var content: List[T] = List()):
  def isEmpty: Boolean = content.isEmpty
  def push(v: T): Unit = content = v :: content
  def pop: T =
    val res = content.head
    content = content.tail
    res

def eval(expr: Expr): Int =
  var exprStack = Stack[Expr]()
  var resStack = Stack[Int]()
  var pcStack = Stack[Int]()
  var expr0 = expr
  var pc = 1

  while !(pcStack.isEmpty && pc == 4) do
    pc match
      case 1 =>
        expr0 match
          case Const(i) =>
            resStack.push(i)
            pc = 4
          case Minus(e1, e2) =>
            pcStack.push(2)
            exprStack.push(e2)
            expr0 = e1
```

```
          pc = 1
      case 2 =>
        pcStack.push(3)
        expr0 = exprStack.pop
        pc = 1
      case 3 =>
        val v2 = resStack.pop
        val v1 = resStack.pop
        resStack.push(v1 - v2)
        pc = 4
      case 4 =>
        if !pcStack.isEmpty then
          pc = pcStack.pop

  resStack.pop
```

This transformation:

- Eliminates recursion (avoids stack overflow)
- Makes control flow explicit
- Enables custom control strategies
- Used by compilers for code generation

Summary Functional interfaces with imperative implementations provide:

- Performance optimization through caching and mutation
- Maintained correctness via object invariants
- Abstraction of implementation details
- Flexibility in control flow representation

Key principle: Hide effects behind pure interfaces to gain efficiency without sacrificing reasoning capabilities.

2.5 Specifications: From English to Math

User stories Capture needs and goals of users. Use a few words to capture the essence of the project. User, topic, purpose => structure Who, Where, What. Example: As a bussiness analyst, when visiting the online dashboard, I want to be able to retrive aggregate visitor statistics over the last day, week, and month.

Requirements Say what the user wants (complete description).

Functional requirements: Concrete testable objectives => "The find function must return a boolean indicating"

Non-functional requirements: General properties. => "The API should respond quickly"

Specifications Say what the program does (Unambiguous functional requirements). Must only covers functional requirements, can include functionality performance, error handling, availability, ... Is Unambiguous. Requirements are for customers. Specifications are for enginners. Example: IETF RFCs, Language specs.

Formal Specifications Tie it all together with code and math. Like a specification but written in a restricted, unambiguous language (Math or code, support formal proofs). Formal specs are for engineers and **computers**.

2.6 Proving

2.6.1 Proof by Induction

Start with a base case. By induction, prove that $n+1$ holds assuming n holds.

Theorem 2.1. *For all $n \in \mathbb{N}$, the sum of the first n natural numbers is:*

$$S(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Proof. We prove this theorem using mathematical induction.

Base Case: For $n = 1$,

$$S(1) = \sum_{i=1}^{1} i = 1$$

The formula gives:

$$\frac{1(1+1)}{2} = \frac{2}{2} = 1$$

Thus, the base case holds.

Inductive Step: Assume the formula holds for some arbitrary $k \in \mathbb{N}$:

$$S(k) = \sum_{i=1}^{k} i = \frac{k(k+1)}{2}$$

We show it also holds for $k+1$:

$$S(k+1) = \sum_{i=1}^{k+1} i = S(k) + (k+1)$$

By the inductive hypothesis:

$$S(k+1) = \frac{k(k+1)}{2} + (k+1)$$

Simplifying:

$$S(k+1) = \frac{k(k+1)}{2} + \frac{2(k+1)}{2} = \frac{k(k+1) + 2(k+1)}{2}$$

Factoring out $(k+1)$:

$$S(k+1) = \frac{(k+1)(k+2)}{2}$$

This matches the formula for $n = k+1$:

$$S(k+1) = \frac{(k+1)((k+1)+1)}{2}$$

□

2.6.2 Proof about Functions

Theorem 2.2. *Consider the Scala function:*

```
def sum(n: Int): Int =
  if n == 0 then 1
  else n + sum(n - 1)
```

Proof. Proof by induction on the Scala function.

Base Case: For $n = 0$,

```
sum(0)
=
if 0 == 0 then 1
  else 0 + sum(0 - 1)
= 1
```

Inductive Step: For $n \geq 0$,

```
sum(n+1)
=
if (n+1) == 0 then 1
  else (n+1) + sum((n+1) - 1)
= // n + 1 >= 1
if false then 1
  else (n+1) + sum((n+1) - 1)
=
(n+1) + sum((n+1) - 1)
=
(n+1) + sum(n)
= // by inductive hypothesis
(n+1) + (n(n + 1))/2
=
(n + 1)(n + 2)/2
=
((n + 1)((n + 1) + 1))/2
```

□

Functions in Programs vs. Mathematical Functions

Stack Overflow Stacks have limited size. Each function call uses stack space. Deep recursion can lead to stack overflow.

Integer Overflow In many programming languages, including Scala, integers have fixed size (e.g., 32-bit or 64-bit). When an operation produces a result exceeding this size, it wraps around to the minimum value, causing incorrect results.

Approaches for Correct Reasoning Solutions:

- Use a precise model of machine integers: map arithmetic operations to:

$$\{\, x \in \mathbb{Z} \mid -2^{31} \leq x \leq 2^{31} - 1 \,\} \pmod{2^{32}}$$

- Check that values are not too large, so the result matches mathematical computation
- Use unbounded integers (e.g., `int` in Python, `BigInt` in Scala)

2.6.3 Examples on Lists

Length

```
extension (xs: List)
  def length: BigInt = xs match
    case Nil => 0            // Nil case
    case Cons(h, t) => 1 + t.length  // Cons case
```

Defining equations:

```
Nil.length = 0               // Nil case
(h :: t).length = 1 + t.length  // Cons case
```

Evaluation Using Substitution Model

```
(42 :: (69 :: Nil)).length
= // Cons case
1 + (69 :: Nil).length
= // Cons case
1 + (1 + Nil.length)
= // Nil case
1 + (1 + 0)
= 2
```

Symbolic Execution Let x, y be arbitrary values. Using the same steps:

```
(x :: (y :: Nil)).length
= // Cons case
1 + (y :: Nil).length
= // Cons case
1 + (1 + Nil.length)
= // Nil case
1 + (1 + 0)
= 2
```

2.6.4 Valid Equations for Use in Proofs

1. The defining equations for our functions (e.g., `length`) for each case
2. **Reflexivity:** $E = E$
3. **Symmetry:** if $E = F$ then $F = E$
4. **Transitivity:** if $E = F$ and $F = G$, then $E = G$ (chaining: $E = F = G$)
5. **Instantiation:** if $A = B$ holds for all values of symbolic x, then $A[C/x] = B[C/x]$ where C is any expression denoting a value
   Example: if $\text{length}(x :: (y :: \text{Nil})) = 2$ then $\text{length}(42 :: (y :: \text{Nil})) = 2$
6. **Substitution:** if $E = F$ and $A = B$ then also $A[F/E] = B[F/E]$
   Example: if $f(x) = x + 1$ and $3 + f(x) = 3 + f(x)$, then $3 + f(x) = 3 + (x + 1)$
7. Results of composing above rules using symbolic execution and other strategies
8. Equations proven using structural induction, using above rules in base and inductive steps

2.6.5 Automated Proof Checking and Search

Proof Checking Proof assistants for interactive theorem proving:

1. Rocq prover: https://rocq-prover.org/
2. HOL prover: https://hol-theorem-prover.org/

3. Isabelle: https://isabelle.in.tum.de/
4. Lean proof assistant: https://lean-lang.org/
5. Lisa proof framework: https://github.com/epfl-lara/lisa

First-Order Logic Provers Based on first-order logic with equality. Examples include:

- E
- SPASS
- Vampire

Proof formats, challenges, and competitions: https://www.tptp.org/

SMT Solvers Satisfiability Modulo Theories solvers build on SAT solvers and extend them with specialized algorithms for:

- Linear arithmetic (Simplex)
- Non-linear arithmetic
- Equality
- Theories of arrays, case classes, and strings

Examples:

- z3
- cvc5
- Princess (written in Scala)

Proof formats, challenges, and competitions: https://smt-lib.org/

Program Verifiers Program verifiers use SMT solvers to automatically prove properties of programs. Examples:

- Stainless verifier for Scala: https://github.com/epfl-lara/stainless/
- Dafny verifier for Dafny language (used at Amazon): https://dafny.org/
- Liquid Haskell verifier for Haskell: https://ucsd-progsys.github.io/liquidhaskell/
- Verus verifier for Rust: https://github.com/verus-lang/verus

Note Proof assistants can also be used to implement provers and verify programs.

2.6.6 Formal Verification

Formal verification uses automated theorem proving and program transformation to construct computer-checked proofs of program correctness. Unlike testing, which checks program behavior on a finite set of inputs, verification can prove properties hold for **all possible inputs**.

2.6.7 Motivation: Limitations of Testing

The Testing Challenge Consider testing commutativity of addition for `Long` integers:

```
assert(x + y == y + x)
```

Testing all cases requires $2^{64} \times 2^{64} = 2^{128} > 10^{38}$ tests. At 10 billion tests per nanosecond, exhaustive testing would take approximately 10^{20} years—ten billion times the age of the universe.

For unbounded integers (`BigInt`), there are infinitely many values to test.

Beyond Testing Testing and fuzzing provide valuable confidence but have fundamental limitations:

- Check behavior only on a tiny fraction of possible executions
- Cannot prove absence of bugs, only detect their presence
- May miss edge cases and rare conditions

ScalaCheck with 5 million tests might pass, while a verifier finds counterexamples:

```
def mSortOK = Prop.forAll { (l: List[Int]) =>
  val res = mSort(l)
  res != Cons(123456, Nil())
}.check(tests(5_000_000))
// + OK, passed 5000000 tests.
```

But Stainless verifier finds:

```
}.ensuring(_ => mSort(lst) != Cons(123456, Nil()))
// [Warning] Found counter-example:
// [Warning] lst: List[Int] -> Cons[Int](123456, Nil[Int]())
```

2.6.8 Formal Verification Overview

Definition Formal verification rigorously proves that computer systems satisfy their specifications by:

1. Defining mathematically rigorous notions of systems satisfying specifications
2. Using automated tools combined with human effort to construct proofs
3. Covering all possible behaviors, not just samples

Verification Workflow A program verifier consists of:

- **Verification condition generator**: Translates programs and specifications into mathematical formulas
- **Theorem prover**: Proves validity of verification conditions

Compiler	Verifier
program → machine code	(program + specification) → formula

If the verification condition is a valid formula, then the program satisfies its specification.

Verification Outcomes Unlike testing (which outputs "program is wrong" or "we don't know"), verifiers can produce three outcomes:

1. **Program is wrong**: Counterexample found
2. **We don't know**: Verification times out or fails

3. **Program is correct**: Proof successfully constructed

2.6.9 Stainless Verifier

Stainless is an open-source verification tool for Scala programs developed at EPFL.

Installation and Usage **Repository:** https://github.com/epfl-lara/stainless/

Documentation: https://epfl-lara.github.io/stainless/installation.html

Running Stainless:

```
stainless fileName.scala ...
```

Running with scala-cli:

```
stainless-cli fileName.scala ...
```

Stainless programs are valid Scala programs and come with a small standard library defining verification constructs and simplified data structures.

Built-in Knowledge Modern verifiers have built-in mathematical knowledge:

- Theorems about integers modulo 2^{64}
- Properties of unbounded integers
- Logical rules from formal mathematical logic
- Automated theorem proving procedures for proof search

For example, Stainless automatically knows `x + y == y + x` without testing.

2.6.10 Specifications with `require` and `ensuring`

Postconditions with `ensuring` The `ensuring` clause specifies properties that must hold for function results:

```
enum List[T]:
  case Nil()
  case Cons(head: T, tail: List[T])

  def size: BigInt =
    this match
      case Nil() => BigInt(0)
      case Cons(_, tail) => BigInt(1) + tail.size
  .ensuring(_ >= 0)  // Size is always non-negative
```

Preconditions with `require` The `require` clause restricts valid function inputs:

```
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] =
  require(xs.size <= ys.size)
  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
      Cons((x, y), zip(xs0, ys0))
    case _ => Nil()
.ensuring(_.map(_._1) == xs)
// Verification succeeds
```

Inductive Reasoning: When proving `ensuring` for `zip(xs, ys)`, Stainless assumes `ensuring` holds for the recursive call `zip(xs0, ys0)`.

Counterexample Detection Without proper preconditions, Stainless finds violations:

```
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] =
  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
      Cons((x, y), zip(xs0, ys0))
    case _ => Nil()
.ensuring(_.map(_._1) == xs)

// [Warning] Found counter-example:
// [Warning]   xs: List[Int] -> Cons[Int](0, Nil[Int]())
// [Warning]   ys: List[Boolean] -> Nil[Boolean]()
```

Restricting Function Calls `require` prevents calling functions with invalid arguments:

```
val exampleCall = zip(Cons(1, Nil()), Nil())
// [Warning] size[Int](Cons[Int](1, Nil[Int]())) <= size[Boolean](Nil[Boolean]())
// [Warning] zip.scala:32:19: => INVALID
```

More Permissive Specifications Use implication instead of `require` for more flexible specifications:

```
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] =
  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
      Cons((x, y), zip(xs0, ys0))
    case _ => Nil()
.ensuring: res =>
  (!(xs.size <= ys.size) || res.map(_._1) == xs) &&
  (!(ys.size <= xs.size) || res.map(_._2) == ys)
```

This specification:

- Uses `==>` written as `!p || q`
- Combines two specifications with `&&`
- Allows calling `zip` without restrictions
- Provides different guarantees depending on list lengths

2.6.11 Essential Uses of `require`

Partial Functions Some functions are undefined for certain inputs and require preconditions:

```
extension[T] (lst: List[T])
  def head: T =
    require(lst != Nil())
    lst match  // No warning for Nil case!
      case Cons(h, t) => h

  def apply(n: BigInt): T =
    require(0 <= n && n < lst.size)
    lst match  // No warning - Stainless proves lst != Nil
      case Cons(h, t) =>
        if n == 0 then h
        else t.apply(n - 1)

val testApplyOK = Cons(1, Cons(2, Cons(3, Nil()))).apply(2)  // Accepted
// val testApplyNo = Cons(1, Cons(2, Cons(3, Nil()))).apply(3)  // Rejected
```

Key point: Stainless uses preconditions to prove pattern match exhaustiveness.

Comparison with Type Checker The Scala type checker alone cannot verify preconditions:

```
def apply(n: BigInt): T =
  require(0 <= n && n < lst.size)  // Ignored by type checker
  lst match
    case Cons(h, t) =>
      if n == 0 then h
      else t.apply(n - 1)

val testApplyOK = Cons(1, Cons(2, Cons(3, Nil()))).apply(2)  // Accepted
val testApplyNo = Cons(1, Cons(2, Cons(3, Nil()))).apply(3)  // Accepted, crashes!

// [warn] match may not be exhaustive.  // But it IS exhaustive!
// [warn] It would fail on pattern case: List.Nil()
```

The compiler warns about non-exhaustive patterns when they're actually exhaustive, but doesn't warn about actual crashes from invalid indices.

2.6.12 Verifying Merge Sort

The `merge` Function

```
def merge(l1: List[Int], l2: List[Int]): List[Int] =
  require(isSorted(l1) && isSorted(l2))
  decreases(l1.length + l2.length)
  (l1, l2) match
    case (Cons(x, xs), Cons(y, ys)) =>
      if x <= y then Cons(x, merge(xs, l2))
      else Cons(y, merge(l1, ys))
    case _ => l1 ++ l2
.ensuring: res =>
  isSorted(res) &&
  res.length == l1.length + l2.length &&
  res.content == l1.content ++ l2.content
```

The `split` Function

```
def split(list: List[Int]): (List[Int], List[Int]) =
  decreases(list)
  list match
    case Cons(x1, Cons(x2, xs)) =>
      val (s1, s2) = split(xs)
      (Cons(x1, s1), Cons(x2, s2))
    case _ => (Nil[Int](), list)
.ensuring: res =>
  res._1.size + res._2.size == list.size &&
  res._1.content ++ res._2.content == list.content
```

The `mSort` Function

```
def mSort(list: List[Int]): List[Int] =
  decreases(list.size)
  list match
    case Cons(h1, Cons(h2, rest)) =>
      val (s1, s2) = split(rest)
      merge(mSort(s1), mSort(s2))
    case _ => list
.ensuring: res =>
  isSorted(res) &&
  res.length <= list.length &&  // Only <=, not ==
  res.content.subsetOf(list.content)
```

Helper: `isSorted`

```
def isSorted(list: List[Int]): Boolean =
  decreases(list)
  list match
    case Cons(x1, tail @ Cons(x2, _)) =>
      x1 <= x2 && isSorted(tail)
    case _ => true
```

2.6.13 Advanced Proofs

Proving List Indexing Properties Some proofs require inductive reasoning on multiple parameters:

```
def appendIndex[T](l1: List[T], l2: List[T], i: BigInt): Unit =
  require(0 <= i && i < (l1 ++ l2).size)  // Well-definedness
  l1 match
    case Cons(x, xs) if i > 0 =>
      appendIndex[T](xs, l2, i - 1)  // Inductive hypothesis
    case _ => ()  // Base case
.ensuring: _ =>
  (l1 ++ l2)(i) == (if i < l1.size then l1(i) else l2(i - l1.size))
```

Proof strategy: Reduce both `l1` and `i` simultaneously.

Interdependent Specifications Verifying one function may require specifications on related functions:

```
extension[T] (xs: List[T])
  def ++(ys: List[T]): List[T] =
    xs match
      case Nil() => ys
      case Cons(h, t) => Cons(h, t ++ ys)
  .ensuring: res =>
    res.size == xs.size + ys.size
```

Without the `size` postcondition on `++`, the `appendIndex` proof fails because Stainless cannot prove:

```
i - l1.size < l2.size  // Needed for safety
```

from

```
i < (l1 ++ l2).size    // What we know
```

Induction Annotation The `@induct` annotation asks Stainless to generate inductive proofs:

```
import stainless.annotation.*

def single[T](x: T) = Cons(x, Nil[T]())

def examQuestion[T](@induct lst: List[T]): Unit =
.ensuring: _ =>
  val ll: List[List[T]] = lst.map(single)
  ll.flatten == lst
```

Stainless generates pattern matching and recursive calls for the inductive proof.

2.6.14 Equivalence Checking

Comparing Implementations Verify that different implementations produce identical results:

```
def isSortedR(l: List[Int]): Boolean =
  def loop(p: Int, l: List[Int]): Boolean =
    l match
```

```
      case Nil() => true
      case Cons(x, xs) if p <= x => loop(x, xs)
      case _ => false
  if l.isEmpty then true
  else loop(l.head, l.tail)

def isSortedB(l: List[Int]): Boolean =
  if l.isEmpty then true
  else if !l.tail.isEmpty && l.head > l.tail.head then false
  else isSortedB(l.tail)
.ensuring(_ == isSortedR(l))  // Verifies equivalence
```

Equivalence Checking Mode Use Stainless's equivalence checking mode:

```
stainless equiv-sorted.scala --equivchk=true --timeout=3 \
  --comparefuns=isSortedB --models=isSortedR

# Output:
# List of functions equivalent to model EquivSorted.isSortedR:
#   EquivSorted.isSortedB
```

2.6.15 Termination and `decreases`

Why Termination Matters Non-terminating functions can prove anything:

```
def f(x: BigInt): BigInt =
  f(x)
.ensuring(_ => 1 == 2)  // "Proves" false statement!
```

Termination checking:

- Ensures sound reasoning via function induction
- Provides specifications "for free"—programmers need not write termination properties
- Catches common programming errors

Accidental Non-termination Be careful with postconditions that invoke the function:

```
def f(x: BigInt): BigInt =
  x + 1
.ensuring(_ => f(x) == 42 && false)  // Infinite recursion!

f(42)
```

Solution: Use `res =>` to refer to the result:

```
def f(x: BigInt): BigInt =
  x + 1
.ensuring(res => res == x + 1)  // Correct
```

Basic `decreases` Clause For integer expressions:

```
import stainless.annotation.*
import stainless.lang.*

def sum(a: Array[Int], from: Int, to: Int): Int =
  require(0 <= from && from <= to && to <= a.length)
  decreases(to - from)  // Measure must decrease
  if from >= to then 0
  else a(from) + sum(a, from + 1, to)
```

Requirements:

- Precondition must imply $0 \leq$ measure

- In each recursive call, measure must strictly decrease

Verification:

- $0 \leq \texttt{to} - \texttt{from}$ follows from precondition
- $\texttt{to} - (\texttt{from} + 1) < \texttt{to} - \texttt{from}$, so measure decreases

Lexicographic Measures For multiple parameters, use tuples with lexicographic ordering:

```
decreases(p, q)
```

The ordering is defined as:

$$(p, q) > (p', q') \iff p > p' \vee (p = p' \wedge q > q')$$

Example: $(100, 2) > (100, 1) > (99, 123456) > (99, 123455) > \ldots$

General requirement: The measure must be ordered by a well-founded relation (no infinite descending chains).

Structural Measures For recursive data structures, measures represent size:

```
def map[U](f: T => U): List[U] =
  decreases(this)  // List size decreases
  this match
    case Nil() => Nil()
    case Cons(head, tail) => Cons(f(head), tail.map(f))
```

Stainless synthesizes internal size functions for:

- Length of lists
- Number of nodes in trees
- Other recursive structures

Custom measure functions can be defined but must themselves be proven terminating.

Measure Inference Stainless can infer some measures automatically:

```
# Check measure validity
stainless file.scala  # Default: checks measures

# Disable measure checking
stainless file.scala --check-measures=false

# Disable measure inference
stainless file.scala --infer-measures=false

# Disable both
stainless file.scala --infer-measures=false --check-measures=false
```

Limitations: Measure inference struggles with:

- Mutual recursion
- Higher-order functions
- Complex recursion patterns

Catching Non-terminating Code Stainless sometimes detects infinite loops:

```
def map[U](f: T => U): List[U] =
  this match
    case Nil() => Nil()
    case Cons(head, tail) => Cons(f(head), this.map(f))  // Bug!

// [Warning] Function map loops given inputs:
// [Warning]   thiss: List[T] -> Cons[T](T#2, Nil[T]())
// [Warning]   f: (T) => U -> (x$$158: T) => U#0
```

2.6.16 Verifying Imperative Code

Array Search Example

```
def find(a: Array[Int], from: Int, to: Int, x: Int): Int =
  require(0 <= from && from <= to && to <= a.size)

  var i = from
  (while i < to && a(i) != x do
    decreases(to - i)
    i = i + 1
  ).invariant(from <= i && i <= to)

  if i < to then i
  else -1
.ensuring: res =>
  (from <= res && res < to && a(res) == x) ||
  res == -1
```

Problem: This specification allows the constant function returning `-1`!

```
def find(a: Array[Int], from: Int, to: Int, x: Int): Int =
  require(0 <= from && from <= to && to <= a.size)
  -1  // Always return -1
.ensuring: res =>
  (from <= res && res < to && a(res) == x) ||
  res == -1
// Verifies!
```

Full Functional Specification Define a specification function:

```
def existsIn(a: Array[Int], from: Int, to: Int, x: Int): Boolean =
  require(0 <= from && from <= to && to <= a.size)
  decreases(to - from)
  !(from == to) &&
  ((a(to - 1) == x) || existsIn(a, from, to - 1, x))
```

Use it in the complete specification:

```
def find(a: Array[Int], from: Int, to: Int, x: Int): Int =
  require(0 <= from && from <= to && to <= a.size)

  var i = from
  (while i < to && a(i) != x do
    decreases(to - i)
    i = i + 1
  ).invariant(from <= i && i <= to && !existsIn(a, from, i, x))

  if i < to then i
  else -1
.ensuring: res =>
  (from <= res && res < to && a(res) == x) ||
  (res == -1 && !existsIn(a, from, to, x))
```

This specification completely characterizes the output: either an index where `x` exists, or `-1` if `x` doesn't exist in the range.

2.6.17 Advanced Example: Balanced Trees

Verification of complex data structures with performance guarantees:

```
def ++(ys: Conc[T]): Conc[T] =
  require(xs.isBalanced && ys.isBalanced)
  decreases(abs(xs.height - ys.height))
  ...
.ensuring: res =>
  appendAssocInst(xs, ys) &&  // Lemma instantiation
  res.isBalanced &&
  res.height <= max(xs.height, ys.height) + 1 &&
  res.height >= max(xs.height, ys.height) &&
  res.toList == xs.toList ++ ys.toList
```

Strengthening specifications: Sometimes specifications must be strengthened beyond the main goal to enable proof.

Static Checks Import Disable runtime checking for verified code:

```
import stainless.lang.StaticChecks.*
```

Without this import, runtime checks for `require`, `ensuring`, and `assert` would make tree operations worse than $O(\log n)$ due to `toList` and `++` calls.

2.6.18 Demo: Expression Simplifier

Verified constant folding with soundness guarantees:

```
def constfold1(e: Expr)(using anyCtx: Env) =
  e match
    case Add(Number(n1), Number(n2)) => Number(n1 + n2)
    case Minus(Number(n1), Number(n2)) => Number(n1 - n2)
    case e => e
.ensuring(evaluate(_) == evaluate(e))

val constFold1Simp = new SoundSimplifier:
  override def apply(e: Expr, anyCtx: Env) =
    constfold1(e)(using anyCtx)

def mapExpr(e: Expr, f: SoundSimplifier)(using anyCtx: Env): Expr =
  val mapped: Expr = e match
    case Number(_) => e
    case Var(_) => e
    case Add(e1, e2) => Add(mapExpr(e1, f), mapExpr(e2, f))
    case Minus(e1, e2) => Minus(mapExpr(e1, f), mapExpr(e2, f))
  f(mapped, anyCtx)
.ensuring(evaluate(_) == evaluate(e))
```

2.6.19 Limitations of Stainless

Current Restrictions

- No support for Scala standard library; Stainless library is small
- Co-variant data structures work but verification is slower
- No sharing of mutable structures (more restrictive than Rust)
- Function values cannot refer to mutable state

- Difficulties with nested arrays (aliasing restrictions, solver performance)
- Function equality is not extensional
- Measure inference doesn't work on instantiated generic types (e.g., `List[List[Int]]`)
- Not all Scala type system features supported (type members, intersections, unions)
- Automatic transformation to functional code can produce confusing error messages

2.6.20 Case Studies

Industrial Applications **European Space Agency (with Ateleris GmbH):**

- ASN.1/ACN Decoders and Encoders—declaratively specified and generated as verified Scala code (VMCAI 2025)
- STIX File System embedded code—from verified Scala to efficient C with preallocated data (NFM 2022)

Data Structure Verification

- Hash tables (LongMap) shown behaviorally equivalent to lists (IJCAR 2024)
- Quite Okay Image Format (https://qoiformat.org/)—proven `decode(encode(img)) == img` (FMCAD 2022)
- Balanced trees and ConcTrees (functional data structures)

Additional Verified Systems

- Tendermint blockchain client
- Algorithms used by Digital Asset
- Soundness of System F (typed λ-calculus with first-class polymorphism)

More examples: https://github.com/epfl-lara/bolts/

2.6.21 Key Takeaways

1. **Verification vs. Testing**: Verification proves properties for all inputs; testing checks finite samples
2. **Specifications**: `require` (preconditions), `ensuring` (postconditions), `decreases` (termination)
3. **Termination**: Essential for sound reasoning; non-terminating functions can "prove" anything
4. **Proof Techniques**: Induction, lemma instantiation, specification strengthening
5. **Practical Use**: Real-world applications in aerospace, blockchain, and critical systems
6. **Trade-offs**: More powerful than testing but requires more effort and has limitations

2.7 Collaborative Software Development

2.7.1 Development Lifecycle

Individual Development Simple cycle: Write code Debug Repeat

Collaborative Development More complex process:

1. Propose feature or fix
2. Debate approach
3. Design solution
4. Implement with tests
5. Code review
6. Document changes
7. Merge to main branch
8. Release
9. Maintain and support

2.7.2 Distributed Workflows

Patch-Based (Email)

```
git format-patch main..feature
git send-email *.patch
# Maintainer applies with: git am
```

Used by: Linux kernel, GCC, GNU projects

Public Forge (GitHub/GitLab)

1. Fork repository
2. Clone, branch, commit, push
3. Open pull request
4. Review and iterate
5. Merge via web UI

Used by: Most open source projects

Custom Forge Internal systems with custom review processes.

Used by: Large companies (AWS, Google, Debian)

2.7.3 Three Aspects of Development

Code Which libraries? What standards? How to test?

People Who can contribute? How to handle conflicts? How to onboard newcomers?

Governance Who decides? How to prioritize? What's the release process?

2.7.4 Collaboration Tools

For Users

- **Release notes:** Document changes between versions
- **Semantic versioning:** `MAJOR.MINOR.PATCH`

- **Bug trackers:** GitHub Issues, Jira

For Developers

- **Code review:** Examine changes before merging
- **CI/CD:** Automated testing and deployment
- **Project boards:** Track tasks and milestones

For Community

- **Code of conduct:** Community standards
- **Contributing guide:** How to contribute
- **Communication:** Mailing lists, chat, forums

2.7.5 Software Ethics

Environmental Energy consumption, e-waste, carbon footprint

Social Privacy, accessibility, online harassment, misinformation

Economic Scams, automation impact, worker conditions

Fairness Software licenses, algorithmic bias, digital rights

Professional Responsibility

- Consider impact of what you build
- Design with empathy
- Prioritize user welfare
- Be transparent and accountable

Be a force for good.

2.8 Monads

A *monad* is a type constructor $F[\_]$ equipped with two operations:

$$\text{pure} : A \to F[A], \qquad \text{flatMap} : F[A] \to (A \to F[B]) \to F[B],$$

satisfying the monad laws:

$$\begin{aligned} \text{(Left identity)} \quad & \text{pure}(a)\ \text{flatMap}\ f = f(a), \\ \text{(Right identity)} \quad & m\ \text{flatMap pure} = m, \\ \text{(Associativity)} \quad & (m\ \text{flatMap}\ f)\ \text{flatMap}\ g = m\ \text{flatMap}\ (x \mapsto f(x)\ \text{flatMap}\ g). \end{aligned}$$

In Scala, any type implementing `map` and `flatMap` with these laws can be used as a monad. The standard example is `List`.

List as a monad.

$$\text{pure}(a) = \texttt{List}(a), \qquad \text{flatMap}(xs, f) = \bigcup_{x \in xs} f(x).$$

Thus `List` models *non-determinism*: a computation may yield several possible results.

Example:

```
val xs = List(1, 2, 3)
val r  = xs.flatMap(x => List(x, x + 1))
// r = List(1, 2, 2, 3, 3, 4)
```

For-comprehension. Scala's `for` syntax desugars to monadic operations:

```
for
  x <- xs
  y <- ys
yield x + y
```

$$\equiv \texttt{xs.flatMap}(x \mapsto \texttt{ys.map}(y \mapsto x + y)).$$

Other monads.

`Option`, `Either`, `Try`, `Future`

all behave as monads, each encoding a computational effect (failure, exceptions, asynchrony, ...).

2.8.1 Queries with For-Expressions

For-expressions in Scala are equivalent to common query languages for databases. They provide a powerful abstraction for working with collections.

Database Example Consider a mini-database of books:

```
case class Book(title: String, authors: List[String])

val books: List[Book] = List(
  Book(title = "Structure and Interpretation of Computer Programs",
       authors = List("Abelson, Harald", "Sussman, Gerald J.")),
  Book(title = "Introduction to Functional Programming",
       authors = List("Bird, Richard", "Wadler, Phil")),
  Book(title = "Effective Java",
       authors = List("Bloch, Joshua")),
  Book(title = "Java Puzzlers",
       authors = List("Bloch, Joshua", "Gafter, Neal")),
  Book(title = "Programming in Scala",
       authors = List("Odersky, Martin", "Spoon, Lex", "Venners, Bill")))
```

Query Examples Find titles of books whose author's name starts with "Bird":

```
for
  b <- books
  a <- b.authors
  if a.startsWith("Bird,")
yield b.title
```

Find all books with "Program" in the title:

```
for
  b <- books
  if b.title.indexOf("Program") >= 0
yield b.title
```

Complex Queries Find names of all authors who have written at least two books:

```
// First attempt - has duplicates
for
  b1 <- books
  b2 <- books
  if b1 != b2
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

This produces duplicates because each pair appears twice (once as (b_1, b_2) and once as (b_2, b_1)). Additionally, if an author has three books, they appear three times (once for each pair of books).

Eliminating Duplicates Use title ordering to ensure pairs appear only once:

```
val repeated =
  for
    b1 <- books
    b2 <- books
    if b1.title < b2.title  // ensures each pair appears once
    a1 <- b1.authors
    a2 <- b2.authors
    if a1 == a2
  yield a1

repeated.distinct  // remove remaining duplicates
```

Better alternative using sets (automatically handles uniqueness):

```
val bookSet = books.toSet
for
  b1 <- bookSet
  b2 <- bookSet
  if b1 != b2
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

2.8.2 Functional Random Generators

For-expressions are not limited to collections. Any type implementing `map` and `flatMap` can use for-expression syntax.

Generator Trait Define a trait for random value generation:

```
trait Generator[+T]:
  self =>  // alias for 'this'
  def generate: T

  def map[S](f: T => S): Generator[S] = new Generator[S]:
    def generate = f(self.generate)

  def flatMap[S](f: T => Generator[S]): Generator[S] = new Generator[S]:
    def generate = f(self.generate).generate
```

Basic Generators

```
val integers = new Generator[Int]:
  val rand = new java.util.Random
  def generate = rand.nextInt()
```

```
val booleans = for x <- integers yield x > 0

def pairs[T, U](t: Generator[T], u: Generator[U]) =
  for
    x <- t
    y <- u
  yield (x, y)
```

Generator Expansion The `booleans` generator expands as follows:

```
    for x <- integers yield x > 0
≡ integers.map(x => x > 0)
≡ new Generator[Boolean] { def generate = integers.generate > 0 }
```

The `pairs` generator expands:

```
    for x <- t; y <- u yield (x, y)
≡ t.flatMap(x => u.map(y => (x, y)))
≡ new Generator[(T, U)] { def generate = (t.generate, u.generate) }
```

Utility Generators

```
def single[T](x: T): Generator[T] = new Generator[T]:
  def generate = x

def range(lo: Int, hi: Int): Generator[Int] =
  for x <- integers yield lo + x.abs % (hi - lo)

def oneOf[T](xs: T*): Generator[T] =
  for idx <- range(0, xs.length) yield xs(idx)
```

Recursive Generators Generate random lists:

```
def lists: Generator[List[Int]] =
  for
    isEmpty <- booleans
    list <- if isEmpty then emptyLists else nonEmptyLists
  yield list

def emptyLists = single(Nil)

def nonEmptyLists =
  for
    head <- integers
    tail <- lists
  yield head :: tail
```

Generate random trees:

```
enum Tree:
  case Inner(left: Tree, right: Tree)
  case Leaf(x: Int)

def trees: Generator[Tree] =
  for
    isLeaf <- booleans
    tree <- if isLeaf then leaves else inners
  yield tree

def leaves = for x <- integers yield Tree.Leaf(x)
```

```
def inners =
  for
    left <- trees
    right <- trees
  yield Tree.Inner(left, right)
```

2.8.3 Property-Based Testing with Generators

Traditional unit testing requires manually creating test inputs. Property-based testing generates random inputs automatically.

Random Test Function

```
def test[T](g: Generator[T], numTimes: Int = 100)
           (test: T => Boolean): Unit =
  for i <- 0 until numTimes do
    val value = g.generate
    assert(test(value), s"test failed for $value")
  println(s"passed $numTimes tests")
```

Example Test

```
test(pairs(lists, lists)): (xs, ys) =>
  (xs ++ ys).length > xs.length  // INCORRECT property!
```

This property fails when `ys` is empty, since `xs.length` $\not>$ `xs.length`.

ScalaCheck ScalaCheck implements this idea with automatic generator derivation:

```
forAll: (l1: List[Int], l2: List[Int]) =>
  l1.size + l2.size == (l1 ++ l2).size  // correct property
```

ScalaCheck provides `given` instances for common types, allowing automatic test data generation based on type signatures.

2.8.4 Monad Laws in Detail

The three monad laws ensure predictable behavior with for-expressions and enable algebraic reasoning about monadic code.

The Three Laws

1. **Associativity:**

$$m.\text{flatMap}(f).\text{flatMap}(g) = m.\text{flatMap}(x \mapsto f(x).\text{flatMap}(g))$$

2. **Left identity (Left unit):**

$$\text{unit}(x).\text{flatMap}(f) = f(x)$$

3. **Right identity (Right unit):**

$$m.\text{flatMap}(\text{unit}) = m$$

Significance for For-Expressions **Associativity** allows inlining nested for-expressions:

```
// These are equivalent
for
  y <- for
    x <- m
    y <- f(x)
  yield y
  z <- g(y)
yield z

// <==> (by associativity)

for
  x <- m
  y <- f(x)
  z <- g(y)
yield z
```

Right unit states:

```
for x <- m yield x  ==  m
```

Left unit has no direct for-expression analogue but ensures `unit` acts as a proper identity element.

Verifying Option is a Monad Definition of `flatMap` for Option:

```
abstract class Option[+T]:
  def flatMap[U](f: T => Option[U]): Option[U] = this match
    case Some(x) => f(x)
    case None => None
```

Left unit law: `Some(x).flatMap(f) == f(x)`

Proof:

$$\begin{aligned} & \texttt{Some(x).flatMap(f)} \\ = \; & \texttt{Some(x) match \{ case Some(x) => f(x) case None => None \}} \\ = \; & \texttt{f(x)} \end{aligned}$$

Right unit law: `opt.flatMap(Some) == opt`

Proof:

$$\begin{aligned} & \texttt{opt.flatMap(Some)} \\ = \; & \texttt{opt match \{ case Some(x) => Some(x) case None => None \}} \\ = \; & \texttt{opt} \end{aligned}$$

Associativity: `opt.flatMap(f).flatMap(g) == opt.flatMap(x => f(x).flatMap(g))`

Proof (case analysis):

Case 1: opt = Some(x)

$$\begin{aligned} & \texttt{opt.flatMap(f).flatMap(g)} \\ = \; & \texttt{f(x).flatMap(g)} \\ = \; & \texttt{opt.flatMap(x => f(x).flatMap(g))} \end{aligned}$$

Case 2: opt = None

$$\begin{aligned} & \texttt{opt.flatMap(f).flatMap(g)} \\ = \; & \texttt{None.flatMap(g)} \\ = \; & \texttt{None} \\ = \; & \texttt{opt.flatMap(x => f(x).flatMap(g))} \end{aligned}$$

Try is NOT a Monad The `Try` type appears monad-like but violates the left unit law:

```
abstract class Try[+T]:
  def flatMap[U](f: T => Try[U]): Try[U] = this match
    case Success(x) =>
      try f(x)
      catch case NonFatal(ex) => Failure(ex)
    case fail: Failure => fail

  def map[U](f: T => U): Try[U] = this match
    case Success(x) => Try(f(x))
    case fail: Failure => fail
```

Left unit violation:

$$\texttt{Try(expr).flatMap(f)} \neq \texttt{f(expr)}$$

The left-hand side will *never* raise a non-fatal exception (it catches them), while the right-hand side *may* raise exceptions from `expr` or `f`.

Trade-off: Monad Laws vs. Useful Properties Try sacrifices the left unit law to gain a more useful property: the **bullet-proof principle**.

Bullet-proof principle: An expression composed from `Try`, `map`, and `flatMap` will never throw a non-fatal exception.

This makes `Try` practical for error handling, even though it's not technically a monad.

2.8.5 Map as Derived Operation

For any monad, `map` can be defined using `flatMap` and `unit`:

$$m.\text{map}(f) = m.\text{flatMap}(x \mapsto \text{unit}(f(x))) = m.\text{flatMap}(f.\text{andThen}(\text{unit}))$$

This shows `map` is derivable and doesn't need to be a primitive operation for monads.

2.8.6 Summary

- For-expressions work with any type defining `map`, `flatMap`, and optionally `withFilter`
- Common monad examples: `List`, `Set`, `Option`, `Generator`, `Future`
- Monads satisfy three laws: associativity, left unit, right unit
- These laws ensure predictable for-expression behavior
- Not all useful types are strict monads (e.g., `Try`), and that's acceptable when the trade-off is worthwhile

2.9 Parallel programming

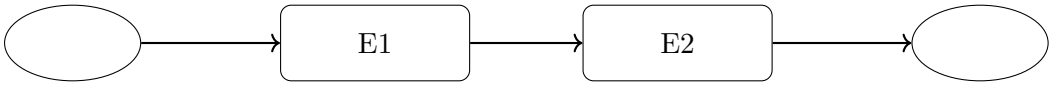
Definition 2.3. Multiple computations happen at once (simultaneously) in physically different (parts of) devices.

Example of physical parallelism:

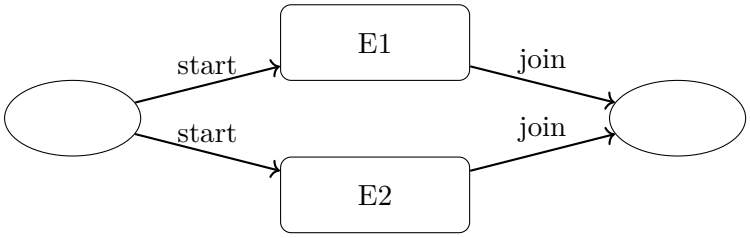
- Several connected computers (cluster, cloud) (Called distributed computing)
- Multiple cores in one chip (multicore CPU) (Most important for the lecture)
- Thousands of cores in a GPU
- Intel® AVX-512 vector instructions that work on 512 bits at once (Meaning 16 floats or 8 doubles at once)
- FPGA (Field Programmable Gate Array) (e.g. AMD® Alveo U200, over 800k LUTs)

2.9.1 Parallel vs Sequential programming

In sequential computation: split tasks into two steps (e1,e2)



In Parallelism: split into two independent tasks (e1,e2), solve in Parallel, then combine (join).



2.9.2 Challenges of parallel programming

Parrallel programming is difficult because of:

- It subsumes sequential programming
- Need to think which of computations are independent
- Different parts of hardware need to communicate
- Its efficiency depends on hardware details more

2.9.3 History context of parallel programming

First established theoretical models were sequential:

- Mathematical computations explained by humans, step by step, as humans have only one mouth, one verbalized train of thoughts
- Turing machine is well-accepted but sequential model of algorithms:
  - read a letter on a tape, write a letter, change the state, repeat
- Our substitution model as a sequential sequence of steps

First physical computers were sequential:

- It was hard enough to build a sequential computer with vaccum tubes

In commercial industry, CPUs were getting faster regularly:

Definition 2.4. Moore's law (Intel): transistors can be made smaller, more fit in a given area

Definition 2.5. Dennard (DRAM) scaling: smaller transistor needs lower voltage, less energy; can switch faster (GHz)

End of Dennard scaling: Early 2005, power density (W/cm^2) stopped decreasing, so clock speed (GHz) stopped increasing.

Multicore Era Instead of making one core faster, the hardware developed:

- More complex cores (pipelined superscalar (Try to automatically parallelize a few consecutive instructions), branch prediction)
- More cores (multicore CPU)

Energy become more important: (mobile devices, sustainability), If the same problem can be solved in parallel, it is more energy efficient to use many cores at lower frequency than one core at high frequency.

- Modern mobile phone: many cores, lower frequency
- Training a LLM: many GPUs, lower frequency

2.9.4 Threads in operating systems

OS sits between hardware and applications. A key role is to handle processes.

- Run user computation (run programs)
- Handle I/O (disk, network, display, keyboard, mouse)

To give CPU resources to multiple applications, OS uses:

- **Preemptive multitasking (Time-slicing):** give a slice of time to each process, then switch to another process
- **Cores:** When CPU has multiple cores, OS can run multiple processes at once

2.9.5 Imperative vs Functional programming for parallelism

Imperative programming Often to synchronize access to shared mutable state, to avoid inconsistencies in non atomics operations. Imperative is programming with mutexes, locks, semaphores, barriers, condition variables... Those are able to cause dangerous bugs like deadlocks, starvation, ...

Functional programming Avoids shared mutable state, so avoids the need for synchronization, Functions compute values instead of writing to variables. Easier to reason about, easier to parallelize. Functional programming is more suitable for parallel programming.

Implicit parallelism Programming language compiler and runtime decide what to run in parallel.

Example: parallel Haskell, various past research projects. Would be wonderful, but not yet efficient.

Explicit parallelism Programmer decides what to run in parallel.

Two important requirements: Result should be correct (same as sequential) and efficient (faster than sequential (To not work for nothing)).

Reduce For parallel reductions, the combine operation must be *associative* if using a collection that preserves order like ParArray or ParVector in Scala. Otherwise, it must be both associative and commutative:

Associativity: The grouping of operations does not matter. Example: addition, $(1 + 2) + 3 = 1 + (2 + 3)$.

Commutativity: The order of operands does not matter. Example: addition, $1 + 2 = 2 + 1$.

Special case: foldLeft and foldRight Unlike `reduce`, `foldLeft` and `foldRight` can work with non-associative and non-commutative operations, but they cannot be parallelized directly because they enforce a specific evaluation order.

To enable parallel execution of fold operations, additional algebraic properties are required:

For foldLeft to be parallelizable: The operation must satisfy: $f(f(a, b), c) = f(f(a, c), b)$

This means the operation must be commutative in its second argument when the first argument is a partial result.

For foldRight to be parallelizable: The operation must satisfy: $f(a, f(b, c)) = f(c, f(b, a))$

This means the operation must be commutative in its first argument when the second argument is a partial result.

In practice: Most operations that satisfy these conditions are already associative and commutative, making `reduce` the preferred choice for parallel computation. Use `fold` with a neutral element when you need both parallelism and a starting value.

2.9.6 Evaluation of parallel programs

Asymptotic analysis Asymptotic analysis studies how the running time or resource usage of a program grows with the input size n. It focuses on the dominant terms that determine performance for large n, ignoring constant factors and lower-order terms.

Big-O notation We say that a function $p(n)$ is in $O(g(n))$ if there exists a constant $M > 0$ and a starting point n_0 such that:

$$p(n) \leq M \cdot g(n) \quad \text{for all } n \geq n_0$$

For example:

$$100n \text{ is } O(n^2) \text{ because } 100n \leq n^2 \text{ for } n \geq 10$$

$$100n \text{ is also } O(n) \text{ because } 100n \leq 100 \cdot n \text{ for all } n \geq 0$$

We often state the *best* $O(f(n))$ we know, but this is not part of the formal definition.

Functions are commonly ordered from slower to faster growth:

$$\log n, \quad n, \quad n \log n, \quad n^2, \quad n^3, \quad 2^n$$

Work and depth We analyze parallel programs with two measures:

- *Work* $W(e)$: number of steps if run sequentially. Treat every `parallel(e1, e2)` as executing both e_1 and e_2; all work must be done.
- *Depth* $D(e)$: number of steps with unbounded parallelism. For `parallel`, take the maximum branch time.

Assume constants so that $D(e) \leq W(e)$.

Composition rules.

$$W(\texttt{parallel}(e_1, e_2)) = W(e_1) + W(e_2) + c_2, \qquad D(\texttt{parallel}(e_1, e_2)) = \max\{D(e_1), D(e_2)\} + c_1.$$

For a call or operation $f(e_1, \ldots, e_n)$:

$$W(f(e_1, \ldots, e_n)) = \sum_{i=1}^{n} W(e_i) + W(f)(v_1, \ldots, v_n),$$

$$D(f(e_1, \ldots, e_n)) = \sum_{i=1}^{n} D(e_i) + D(f)(v_1, \ldots, v_n),$$

where v_i are the values of e_i. For primitive integer operations, $W(f)$ and $D(f)$ are constants.

Key insight. Large depth limits speedup. If depth is high, adding processors yields little benefit.

Example: divide-and-conquer `segmentPar`.

```
def segmentPar(xs: Array[T], p: Double, from: Int, len: Int): Int =
  if len < threshold then sumSegment(xs, p, from, from + len)
  else val (l, r) = parallel(segmentPar(xs, p, from, len/2),
      segmentPar(xs, p, from + len/2, len - len/2))
    l + r
```

Let input length be L (power of two) and leaves do $O(L)$ work when $L <$ `threshold`.

Work.

$$W(L) = \begin{cases} O(L), & L < \texttt{threshold}, \\ 2\,W(L/2) + c, & \text{otherwise}, \end{cases} \qquad \Rightarrow \qquad W(L) = O(L).$$

Depth.

$$D(L) = \begin{cases} O(L), & L < \texttt{threshold}, \\ D(L/2) + c, & \text{otherwise}, \end{cases} \qquad \Rightarrow \qquad D(L) = O(\log L).$$

Even though work stays linear, splitting halves the critical path, giving logarithmic depth.

2.10 Web application

Unlike traditional desktop applications that can be built as monoliths, web applications must be distributed across networks, which fundamentally changes their architecture. This network constraint necessitates a multi-tier approach to handle the separation between client and server.

2.10.1 The 3-Tier Architecture

Web applications typically follow a 3-tier architecture, separating concerns into distinct layers:

- **Data Layer (State):** Manages persistent storage and the application's state
- **Application Layer (Logic):** Contains the business logic and processing rules
- **Presentation Layer:** Handles user interface rendering, typically in the browser

These three layers can be implemented using a single fullstack language (such as JavaScript/TypeScript with Node.js), or divided into separate programs using specialized languages optimized for each layer (e.g., SQL for data, Java/Python for logic, HTML/CSS/JavaScript for presentation).

2.10.2 Best Practices by Layer

Data Layer: The fundamental principle is to avoid storing redundant information in the state. When information can be derived from existing data, it should not be persisted. For example, determining whose turn it is in a game should be computed by comparing the current player ID with the active player ID, rather than storing a separate `isMyTurn` flag. This approach significantly reduces the risk of inconsistent states, as there is only one source of truth. This principle is actually called "normalization" in databases and "single source of truth" in software design.

Application Layer: Given the minimalist approach to the data layer, the application layer takes responsibility for computing derived information. It transforms the raw state into the various forms needed by different parts of the system, ensuring consistency through controlled computation rather than redundant storage.

Presentation Layer: Unlike the data layer, duplication is acceptable in the presentation layer. The same information may be displayed in multiple locations or formats for user experience purposes, as this redundancy does not affect system consistency.

2.11 State Machines

A **finite state machine (FSM)** or **finite automaton** is a mathematical model of computation used to design and analyze systems with discrete states and well-defined transitions.

Formal Definition An FSM is defined as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$:

- Q: finite set of states
- Σ: finite alphabet (set of input symbols)
- $\delta : Q \times \Sigma \to Q$: transition function
- $q_0 \in Q$: initial state
- $F \subseteq Q$: set of accepting (final) states

An FSM processes a string $w = a_1 a_2 \dots a_n$ by starting in q_0 and applying transitions:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{\cdots} q_n$$

The string is **accepted** if $q_n \in F$.

Types of Automata

Deterministic Finite Automaton (DFA) For each state and input symbol, there is exactly one next state. The transition function is total: $\delta : Q \times \Sigma \to Q$.

Nondeterministic Finite Automaton (NFA) A state may have zero, one, or multiple transitions for a given input symbol. The transition function becomes: $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ (returns a set of states).

Mealy Machine Outputs depend on both state and input. Extended transition function: $\delta : Q \times \Sigma \to Q \times \Omega$ where Ω is the output alphabet.

Moore Machine Outputs depend only on the current state. Output function: $\lambda : Q \to \Omega$.

State Diagrams State machines are visualized as directed graphs:

- **Nodes** represent states
- **Edges** represent transitions, labeled with input symbols
- **Initial state** indicated by an incoming arrow from nowhere
- **Accepting states** drawn with double circles

Example: DFA that accepts binary strings ending in "01"

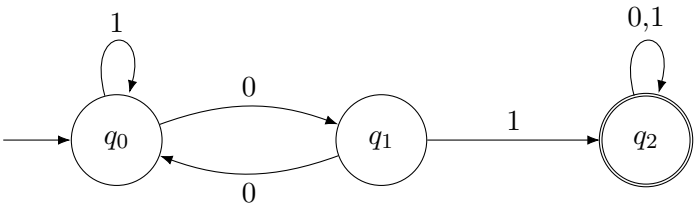


Figure 2: DFA accepting strings ending in "01". Example: "110101" is accepted.

State meanings:

- q_0: haven't seen a 0 recently (or just saw a 1)
- q_1: just saw a 0, waiting for 1
- q_2: saw "01", accepting state (once reached, stay here)

Application: String Matching String matching is a fundamental problem: given a text T of length n and a pattern P of length m, find all occurrences of P in T.

Finite automata provide an elegant solution: construct an FSM that recognizes strings containing P as a substring.

Naive Approach Check every position in T: for each i, compare $T[i..i+m-1]$ with P.

- **Time complexity:** $O(nm)$ in worst case
- **Example worst case:** $T =$ "AAAAAAA...", $P =$ "AAAB"

Knuth-Morris-Pratt (KMP) Algorithm KMP uses a DFA implicitly through a **failure function** to achieve $O(n+m)$ time.

Key insight: When a mismatch occurs, use information from the pattern itself to avoid rechecking characters we already know match.

Failure Function: For pattern $P[0..m-1]$, define $\pi[i]$ as the length of the longest proper prefix of $P[0..i]$ that is also a suffix of $P[0..i]$.

A **proper prefix** of a string is a prefix that is not equal to the string itself.

Example: Pattern "ABABC"

i	0	1	2	3	4
$P[i]$	A	B	A	B	C
$\pi[i]$	0	0	1	2	0

Explanation:

- $\pi[0] = 0$: "A" has no proper prefix
- $\pi[1] = 0$: "AB" – no prefix equals suffix
- $\pi[2] = 1$: "ABA" – prefix "A" = suffix "A"
- $\pi[3] = 2$: "ABAB" – prefix "AB" = suffix "AB"
- $\pi[4] = 0$: "ABABC" – no prefix equals suffix

State Machine Interpretation:

The failure function defines a DFA where:

- State q means "we have matched the first q characters of P"
- On matching input, advance: $q \to q + 1$
- On mismatch, follow failure link: $q \to \pi[q-1]$ (and retry)

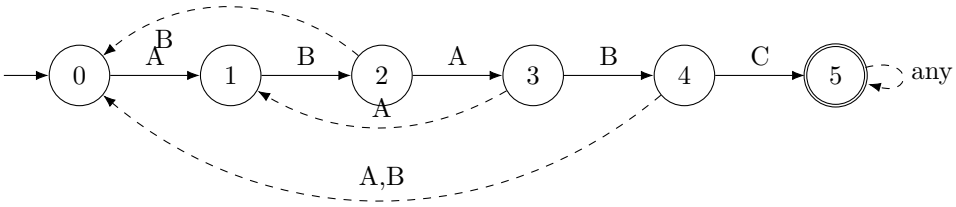


Figure 3: DFA for pattern "ABABC". Solid arrows: successful matches. Dashed: failure transitions.

Algorithm – Computing Failure Function:

```
def compute_failure_function(pattern):
    m = len(pattern)
    pi = [0] * m
    k = 0  # length of previous longest prefix-suffix

    for i in range(1, m):
        # Follow failure links until match or reach start
        while k > 0 and pattern[k] != pattern[i]:
            k = pi[k - 1]

        # Extend match if possible
        if pattern[k] == pattern[i]:
            k += 1

        pi[i] = k

    return pi
```

Example execution for "ABABC":

1. $i = 1$: 'B' $\neq$ 'A' (at $k = 0$), no match $\Rightarrow \pi[1] = 0$
2. $i = 2$: 'A' $=$ 'A' (at $k = 0$), match $\Rightarrow k = 1, \pi[2] = 1$
3. $i = 3$: 'B' $=$ 'B' (at $k = 1$), match $\Rightarrow k = 2, \pi[3] = 2$
4. $i = 4$: 'C' $\neq$ 'A' (at $k = 2$), fail to $k = \pi[1] = 0$; 'C' $\neq$ 'A' $\Rightarrow \pi[4] = 0$

Algorithm – KMP Search:

```
def kmp_search(text, pattern):
    n = len(text)
    m = len(pattern)
    pi = compute_failure_function(pattern)
    matches = []
    q = 0  # number of characters matched

    for i in range(n):
        # Follow failure function on mismatch
        while q > 0 and pattern[q] != text[i]:
            q = pi[q - 1]

        # Extend match
        if pattern[q] == text[i]:
            q += 1

        # Complete match found
        if q == m:
            matches.append(i - m + 1)  # starting position
            q = pi[q - 1]  # continue searching

    return matches
```

Example trace for text "ABABDABACDABABCABABA", pattern "ABABC":

1. Scan "ABABD": matches up to "ABAB", then 'D' $\neq$ 'C'. Failure function: fall back to $q = 2$ (we still have "AB" matched).
2. Continue from position 5: "ABACD"... no match.
3. At position 10: "ABABC" – **match found at index 10**.
4. Continue scanning... (process continues)

Complexity Analysis:

- **Preprocessing:** $O(m)$ to compute π
- **Matching:** $O(n)$ – each character examined at most twice (once advancing i, once following failure links)
- **Total:** $O(n + m)$

Correctness intuition: The failure function ensures that when we fail at position q, we already know that the text matches the pattern for the previous $\pi[q-1]$ characters. We can safely skip re-checking them.

Comparison: Naive vs. KMP

Algorithm	Preprocessing	Matching
Naive	$O(1)$	$O(nm)$
KMP	$O(m)$	$O(n)$

For $n = 10^6$ and $m = 100$:

- Naive worst case: 10^8 operations
- KMP: $\sim 10^6$ operations (100Œ faster)

Other String Matching Algorithms

Boyer-Moore Scans pattern from right to left. Often faster in practice due to large skip distances. Average case: $O(n/m)$.

Rabin-Karp Uses hashing. Computes hash of pattern and compares with rolling hash of text substrings. $O(n + m)$ expected time, $O(nm)$ worst case.

Aho-Corasick Extends KMP to match multiple patterns simultaneously using a trie structure. Used in tools like `grep -F`. $O(n + m + z)$ where z is the number of matches.

Practical Applications of FSMs

Lexical analysis Tokenizing source code in compilers. Regular expressions are compiled to DFAs for pattern matching.

Network protocols TCP state machine: `CLOSED`, `LISTEN`, `SYN_SENT`, `ESTABLISHED`, `FIN_WAIT`, etc.

Text editors Syntax highlighting uses FSMs to identify keywords, strings, comments in real-time.

Game development Character AI: `Idle` → `Patrol` → `Chase` → `Attack` states.

Hardware design Digital circuits, vending machines, traffic light controllers.

UI workflows Multi-step forms: `Input` → `Validate` → `Confirm` → `Complete`.

Design Principles When designing state machines:

1. **Minimize states:** Each state should represent a distinct, meaningful situation
2. **Define all transitions:** What happens for every input in every state?
3. **Identify invariants:** What properties must hold in each state?
4. **Avoid state explosion:** Use hierarchical state machines for complex systems
5. **Document:** State diagrams are worth a thousand lines of code

Common pitfall: Mixing state with behavior. Keep state simple (data) and behavior separate (transition functions).

2.12 Relational Algebra: Theoretical Foundation

Relational algebra is a procedural query language that consists of a set of operations on relations. A relation is a set of tuples, where each tuple represents a row and has attributes (columns). Relational algebra operations are closed: they take relations as input and produce relations as output.

2.12.1 Declarative vs Procedural Implementations

Relational algebra serves as the theoretical foundation for both declarative and procedural query languages:

Declarative: SQL SQL is a **declarative** language: you specify *what* you want, not *how* to get it. The database query optimizer translates SQL into relational algebra operations and determines the execution plan.

Example:

```
SELECT name, email
FROM students
WHERE age > 18;
```

The user describes the desired result. The database engine decides whether to:

- Use an index on the `age` column
- Perform a sequential scan
- Apply selection before or after projection

Procedural: Collection APIs Scala collections, Java Streams, and C# LINQ are **procedural**: you explicitly chain operations in a specific order, defining the execution strategy.

Example (Scala):

```
students
  .filter(_.age > 18)
  .map(s => (s.name, s.email))
```

The programmer explicitly specifies:

1. First, filter the collection (selection)
2. Then, transform each element (projection)

The execution order is deterministic and controlled by the programmer, not an optimizer.

Hybrid Approach: LINQ in C# C# LINQ offers both approaches:

Query Syntax (Declarative):

```
from s in students
where s.Age > 18
select new { s.Name, s.Email }
```

Method Syntax (Procedural):

```
students
  .Where(s => s.Age > 18)
  .Select(s => new { s.Name, s.Email })
```

Both compile to the same intermediate representation, but the query syntax mimics SQL's declarative style.

Key Differences

Aspect	SQL (Declarative)	Collections (Procedural)
Optimization	Automatic by query planner	Manual by programmer
Execution order	Determined by optimizer	Explicit in code
Physical access	Abstracted away	Iterator-based
Parallelization	Transparent	Explicit (e.g., `.par`)
Debugging	Requires query analysis tools	Standard debugger

2.12.2 Basic Operations

Selection (σ) The selection operation filters tuples based on a predicate condition.

$$\sigma_{\text{condition}}(R)$$

Returns all tuples from relation R that satisfy the condition.

Mathematical Example: $\sigma_{\text{age}>18}(\text{Students})$ returns all students older than 18.

SQL:

```sql
SELECT *
FROM students
WHERE age > 18;
```

Scala:

```scala
students.filter(_.age > 18)
```

Projection (π) The projection operation selects specific attributes from a relation, eliminating duplicates.

$$\pi_{\text{attr}_1,\text{attr}_2,\ldots}(R)$$

Returns a relation containing only the specified attributes.

Mathematical Example: $\pi_{\text{name,email}}(\text{Students})$ returns only names and emails.

SQL:

```sql
SELECT DISTINCT name, email
FROM students;
```

Scala:

```scala
students.map(s => (s.name, s.email)).distinct
```

Note: The projection operation π in relational algebra automatically eliminates duplicates (returns a set). In SQL, use `DISTINCT` to enforce this; in Scala, use `.distinct` or work with `Set` instead of `List`.

Cartesian Product ($\times$) The Cartesian product combines every tuple from the first relation with every tuple from the second relation.

$$R \times S$$

If R has n tuples and S has m tuples, $R \times S$ has $n \times m$ tuples. If R has k attributes and S has l attributes, $R \times S$ has $k + l$ attributes.

Mathematical Example:

$$\begin{aligned} R &= \{(1, \text{Alice}), (2, \text{Bob})\} \\ S &= \{(\text{Math}, 90), (\text{CS}, 85)\} \\ R \times S &= \{(1, \text{Alice}, \text{Math}, 90), (1, \text{Alice}, \text{CS}, 85), \\ &\quad (2, \text{Bob}, \text{Math}, 90), (2, \text{Bob}, \text{CS}, 85)\} \end{aligned}$$

SQL:

```sql
SELECT *
FROM students, courses;
-- Or explicitly:
SELECT *
FROM students CROSS JOIN courses;
```

Scala: The Cartesian product can be implemented using `flatMap` or for-comprehensions:

Using flatMap:

```
students.flatMap(s => courses.map(c => (s, c)))
```

Using for-comprehension:

```
for {
  s <- students
  c <- courses
} yield (s, c)
```

Both approaches produce the same result: every student paired with every course. The for-comprehension is syntactic sugar that desugars to the `flatMap` version.

Union ($\cup$) The union operation combines tuples from two relations, eliminating duplicates.

$$R \cup S$$

Requirement: R and S must be union-compatible (same number of attributes with compatible types).

Mathematical Example:

$$\begin{aligned} \text{CS\_Students} &= \{(\text{Alice}, 1), (\text{Bob}, 2)\} \\ \text{Math\_Students} &= \{(\text{Bob}, 2), (\text{Charlie}, 3)\} \\ \text{CS\_Students} \cup \text{Math\_Students} &= \{(\text{Alice}, 1), (\text{Bob}, 2), (\text{Charlie}, 3)\} \end{aligned}$$

SQL:

```
SELECT student_id, name FROM cs_students
UNION
SELECT student_id, name FROM math_students;
```

Scala:

```
(csStudents ++ mathStudents).distinct
// Or using union (for sets):
csStudents.toSet.union(mathStudents.toSet)
```

Set Difference ($-$) The set difference returns tuples in the first relation but not in the second.

$$R - S$$

Requirement: R and S must be union-compatible.

Mathematical Example:

$$\text{CS\_Students} - \text{Math\_Students} = \{(\text{Alice}, 1)\}$$

SQL:

```
SELECT student_id, name FROM cs_students
EXCEPT
SELECT student_id, name FROM math_students;
```

Scala:

```
csStudents.diff(mathStudents)
// Or for sets:
csStudents.toSet.diff(mathStudents.toSet)
```

Rename (ρ) The rename operation changes the name of a relation or its attributes.

$$\rho_{S(A_1,A_2,\ldots)}(R)$$

Renames relation R to S and its attributes to $A_1, A_2, \ldots$

SQL:

```sql
SELECT name AS student_name, age AS student_age
FROM students;
```

Scala:

```scala
// Renaming is implicit through mapping to new structures
students.map(s => StudentRenamed(s.name, s.age))
```

2.12.3 Derived Operations

Intersection ($\cap$) The intersection returns tuples that appear in both relations.

$$R \cap S = R - (R - S)$$

Mathematical Example:

$$\text{CS\_Students} \cap \text{Math\_Students} = \{(\text{Bob}, 2)\}$$

SQL:

```sql
SELECT student_id, name FROM cs_students
INTERSECT
SELECT student_id, name FROM math_students;
```

Scala:

```scala
csStudents.intersect(mathStudents)
// Or for sets:
csStudents.toSet.intersect(mathStudents.toSet)
```

Natural Join ($\bowtie$) The natural join combines tuples from two relations based on common attributes with equal values.

$$R \bowtie S = \pi_{\text{all attributes}}(\sigma_{R.A=S.A}(R \times S))$$

where A represents the common attributes.

Mathematical Example:

$$\begin{aligned} \text{Students} &= \{(1, \text{Alice}), (2, \text{Bob})\} \\ \text{Grades} &= \{(1, 90), (2, 85)\} \\ \text{Students} \bowtie \text{Grades} &= \{(1, \text{Alice}, 90), (2, \text{Bob}, 85)\} \end{aligned}$$

SQL:

```sql
SELECT s.student_id, s.name, g.grade
FROM students s
JOIN grades g ON s.student_id = g.student_id;
```

Scala:

```scala
// Using for-comprehension (equivalent to flatMap + filter + map):
for {
  s <- students
  g <- grades
  if s.studentId == g.studentId
} yield (s.studentId, s.name, g.grade)

// Or using flatMap explicitly:
students.flatMap(s =>
  grades
    .filter(g => g.studentId == s.studentId)
    .map(g => (s.studentId, s.name, g.grade))
)
```

Theta Join ($\bowtie_\theta$) A generalized join with an arbitrary condition θ.

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

Mathematical Example: Students $\bowtie_{\text{Students.age}>\text{Courses.min\_age}}$ Courses

SQL:

```sql
SELECT *
FROM students s, courses c
WHERE s.age > c.min_age;
```

Scala:

```scala
for {
  s <- students
  c <- courses
  if s.age > c.minAge
} yield (s, c)
```

Division ($\div$) The division operation finds tuples in one relation that are associated with all tuples in another relation.

$$R \div S$$

Returns tuples from R that match with every tuple in S.

Mathematical Example: Find students enrolled in all courses:

$$\text{Enrollments}(student\_id, course\_id) \div \text{AllCourses}(course\_id)$$

SQL:

```sql
SELECT e.student_id
FROM enrollments e
GROUP BY e.student_id
HAVING COUNT(DISTINCT e.course_id) = (SELECT COUNT(*) FROM all_courses);
```

Scala:

```scala
val allCourseIds = allCourses.map(_.courseId).toSet
enrollments
  .groupBy(_.studentId)
  .filter { case (_, enrolls) =>
    enrolls.map(_.courseId).toSet == allCourseIds
  }
  .keys
```

2.12.4 Properties

Closure Property All relational algebra operations produce relations, allowing operations to be composed:

$$\pi_{\text{name}}(\sigma_{\text{age}>18}(\text{Students} \bowtie \text{Grades}))$$

Scala:

```
students
  .flatMap(s => grades.filter(_.studentId == s.studentId).map(g => (s, g)))
  .filter { case (s, _) => s.age > 18 }
  .map { case (s, _) => s.name }
```

Commutativity

- $R \cup S = S \cup R$
- $R \cap S = S \cap R$
- $R \times S \neq S \times R$ (attributes ordered differently)
- $R \bowtie S = S \bowtie R$ (for natural join)

Associativity

- $(R \cup S) \cup T = R \cup (S \cup T)$
- $(R \cap S) \cap T = R \cap (S \cap T)$
- $(R \times S) \times T = R \times (S \times T)$

Equivalence Rules Multiple algebraic expressions can represent the same query:

$$\sigma_{c_1 \wedge c_2}(R) = \sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_2}(\sigma_{c_1}(R))$$

These equivalences enable query optimization in database systems.

2.12.5 Complex Queries

Eliminating Duplicate Pairs with Ordering When querying pairs of elements from the same collection, using inequality (`!=`) produces each pair twice: (a, b) and (b, a). Using ordering comparison eliminates this symmetry.

Problem: Finding authors with multiple books

```
// First attempt - produces duplicates
for
  b1 <- books
  b2 <- books
  if b1 != b2  // Each pair appears twice: (b1, b2) and (b2, b1)
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

Solution: Use ordering on a unique attribute

```
// Better: use < on titles to ensure each pair appears once
for
  b1 <- books
  b2 <- books
  if b1.title < b2.title  // Only keeps pairs where title1 < title2
```

```
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

This works because:

- String comparison is transitive: if $a < b$ then $b \not< a$
- Each unordered pair $\{b_1, b_2\}$ appears exactly once as the ordered pair (b_1, b_2) where b_1.title $< b_2$.title
- Requires a unique, orderable attribute (like ID, title, etc.)

General pattern:

```
// For any collection with unique IDs
for
  x <- collection
  y <- collection
  if x.id < y.id  // Eliminates symmetric pairs
  // ... rest of query
yield result
```

Alternative: Convert to Set

```
val bookSet = books.toSet
for
  b1 <- bookSet
  b2 <- bookSet
  if b1 != b2  // Set operations are more efficient
  a1 <- b1.authors
  a2 <- b2.authors
  if a1 == a2
yield a1
```

Using `Set` automatically handles uniqueness but doesn't eliminate the (a, b) vs (b, a) duplication issue, so you may still need `.distinct` on the final result.