```
Algebra
==================

Aims:

* Give the students experience working with data classes

* Give the students experience working with operator overloading and
  infix functions

* Give the students experience working with generic classes and
  function objects

* Encourage good testing practices by having a code coverage target
  for tests

Guide to breakdown of marks (out of 10):

Do not give a student an A* if they achieve less than 90% test
coverage for any of their classes. In the PPT session for this lab,
emphasise the importance of thorough testing.

Also do not give a student an A* if their solution features loops
(loops in their tests are fine). This is not in order to give them the
impression that loops are bad. Instead, it is to encourage them to
further explore the functional programming features that Kotlin
offers. Discussing the pros and cons of loops vs.\ a functional
approach will be a good point for discussing during the PPT.

- 2 marks for implementing vectors over doubles

- 3 marks for implementing matrices over doubles

- 3 marks for lifting these vector and matrix implementations to a
  generic form

- 2 marks for the quality of the new tests that are written

Apart from the notes on when not to give an A* (which you should
enforce), this is just a guide - please use your judgement when
deciding how to score the exercise.
```

```kotlin
package algebra.generic

class AlgebraFactory<T>(
    val plus: (T, T) -> T,
    val times: (T, T) -> T,
) {
    fun makeVector(elements: List<T>): Vector<T> = Vector(plus, times, elements)

    fun makeMatrix(rows: List<List<T>>): Matrix<T> = Matrix(plus, times, rows.map {
it -> Vector(plus, times, it) })
}
```

```kotlin
package algebra.generic

import algebra.real.times
import kotlin.IllegalArgumentException
import kotlin.math.max

data class Matrix<T>(
    private val plus: (T, T) -> T,
    private val times: (T, T) -> T,
    private val rows: List<Vector<T>>,
) {

    constructor(
        plus: (T, T) -> T,
        times: (T, T) -> T,
        vararg rows: Vector<T>,
    ) : this(plus, times, rows.toList())

    val numRows = if (rows.isEmpty()) {
        throw IllegalArgumentException()
    } else {
        rows.size
    }
    val numColumns = rows[0].length

    init {
        if (rows.any { it.length != numColumns }) {
            throw IllegalArgumentException()
        }
    }

    operator fun get(rowIndex: Int, columnIndex: Int): T =
        if (rowIndex !in 0..<numRows || columnIndex !in 0..<numColumns) {
            throw IndexOutOfBoundsException()
        } else {
            rows[rowIndex][columnIndex]
        }

    fun getRow(rowIndex: Int): Vector<T> =
        if (rowIndex !in 0..<numRows) {
            throw IndexOutOfBoundsException()
        } else {
            rows[rowIndex]
        }

    fun getColumn(columnIndex: Int): Vector<T> =
        if (columnIndex !in 0..<numColumns) {
            throw IndexOutOfBoundsException()
        } else {
            Vector(plus, times, (0..<numRows).map { rowIndex -> this[rowIndex, columnIndex] })
        }

    operator fun get(rowIndex: Int): Vector<T> = getRow(rowIndex)

    operator fun plus(other: Matrix<T>): Matrix<T> =
        if (numRows != other.numRows || numColumns != other.numColumns) {
            throw UnsupportedOperationException()
        } else {
            copy(
                rows = rows.mapIndexed { rowIndex, row ->
                    row + other.rows[rowIndex]
                },
            )
        }

    operator fun times(other: Matrix<T>): Matrix<T> =
        if (numColumns != other.numRows) {
```

```kotlin
            throw UnsupportedOperationException()
        } else {
            copy(
                rows = rows.map { rowVector ->
                    Vector(
                        plus,
                        times,
                        (0..<other.numColumns).map { columnIndex ->
                            rowVector dot other.getColumn(columnIndex)
                        },
                    )
                },
            )
        }

    operator fun times(scalar: T): Matrix<T> =
        copy(rows = rows.map { row -> row * scalar })

    fun leftMultiply(scalar: T): Matrix<T> =
        copy(rows = rows.map { row -> scalar * row })

    operator fun iterator(): Iterator<Vector<T>> = object : Iterator<Vector<T>> {
        private var index = 0

        override fun hasNext(): Boolean = index < numRows

        override fun next(): Vector<T> = this@Matrix[index++]
    }

    override fun toString(): String {
        val largestColumnEntry: List<Int> = (0..<numColumns).map { column ->
            (0..<numRows).map { row ->
                this[row, column].toString().length
            }.reduce(::max)
        }
        return rows.map { row ->
            "[" + (0..<numColumns).map { column ->
                val entryString = row[column].toString()
                " ".repeat(1 + largestColumnEntry[column] - entryString.length) + entryString
            }.joinToString(separator = "") + " ]"
        }.joinToString(separator = "\n")
    }
}

operator fun <T> T.times(matrix: Matrix<T>) = matrix.leftMultiply(this)
```

```kotlin
package algebra.generic

data class Vector<T>(
    private val plus: (T, T) -> T,
    private val times: (T, T) -> T,
    private val elements: List<T>,
) {

    // Providing a varargs constructor is an extension
    constructor(
        plus: (T, T) -> T,
        times: (T, T) -> T,
        vararg elements: T,
    ) : this(plus, times, elements.toList())

    val length = elements.size

    init {
        if (length <= 0) throw IllegalArgumentException()
    }

    operator fun T.plus(other: T): T = plus(this, other)

    operator fun T.times(other: T): T = times(this, other)

    operator fun get(index: Int): T =
        if (index !in 0..<length) throw IndexOutOfBoundsException() else
elements[index]

    operator fun iterator(): Iterator<T> = object : Iterator<T> {
        private var index = 0

        override fun hasNext(): Boolean = index < length

        override fun next(): T = this@Vector[index++]
    }

    operator fun plus(other: Vector<T>): Vector<T> =
        if (length != other.length) {
            throw UnsupportedOperationException()
        } else {
            copy(elements = elements.mapIndexed { index, element -> element +
other[index] })
        }

    operator fun times(scalar: T): Vector<T> =
        copy(elements = elements.map { it * scalar })

    infix fun dot(other: Vector<T>): T =
        if (length != other.length) {
            throw UnsupportedOperationException()
        } else {
            elements.zip(other.elements).map { (a, b) -> a * b }
                .reduce(plus)
        }

    override fun toString(): String = elements.joinToString(prefix = "(", postfix =
")")

    fun leftMultiply(scalar: T): Vector<T> =
        copy(elements = elements.map { times(scalar, it) })
}

operator fun <T> T.times(vector: Vector<T>): Vector<T> =
    vector.leftMultiply(this)
```

```kotlin
package algebra.real

import kotlin.math.max

data class Matrix(private val rows: List<Vector>) {

    constructor(vararg elements: Vector) : this(elements.toList())

    val numRows = if (rows.isEmpty()) throw IllegalArgumentException() else
rows.size
    val numColumns = rows[0].length

    init {
        if (rows.any { it.length != numColumns }) {
            throw IllegalArgumentException()
        }
    }

    operator fun get(rowIndex: Int, columnIndex: Int): Double =
        if (rowIndex !in 0..<numRows || columnIndex !in 0..<numColumns) {
            throw IndexOutOfBoundsException()
        } else {
            rows[rowIndex][columnIndex]
        }

    fun getRow(rowIndex: Int): Vector =
        if (rowIndex !in 0..<numRows) {
            throw IndexOutOfBoundsException()
        } else {
            rows[rowIndex]
        }

    fun getColumn(columnIndex: Int): Vector =
        if (columnIndex !in 0..<numColumns) {
            throw IndexOutOfBoundsException()
        } else {
            Vector((0..<numRows).map { rowIndex -> this[rowIndex, columnIndex] })
        }

    operator fun get(rowIndex: Int): Vector = getRow(rowIndex)

    operator fun plus(other: Matrix): Matrix =
        if (numRows != other.numRows || numColumns != other.numColumns) {
            throw UnsupportedOperationException()
        } else {
            Matrix(
                rows.mapIndexed { rowIndex, row ->
                    row + other.rows[rowIndex]
                },
            )
        }

    operator fun times(other: Matrix): Matrix =
        if (numColumns != other.numRows) {
            throw UnsupportedOperationException()
        } else {
            Matrix(
                rows.map { rowVector ->
                    Vector(
                        (0..<other.numColumns).map { columnIndex ->
                            rowVector dot other.getColumn(columnIndex)
                        },
                    )
                },
            )
        }

    operator fun times(scalar: Double): Matrix =
        Matrix(rows.map { row -> scalar * row })

    operator fun iterator(): Iterator<Vector> = object : Iterator<Vector> {
        private var index: Int = 0

        override fun hasNext(): Boolean = index < numRows

        override fun next(): Vector = this@Matrix[index++]
    }

    override fun toString(): String {
        val largestColumnEntry: List<Int> = (0..<numColumns).map { column ->
            (0..<numRows).map { row ->
                this[row, column].toString().length
            }.reduce(::max)
        }
        return rows.map { row ->
            "[" + (0..<numColumns).map { column ->
                val entryString = row[column].toString()
                " ".repeat(1 + largestColumnEntry[column] - entryString.length) + entryString
            }.joinToString(separator = "") + " ]"
        }.joinToString(separator = "\n")
    }
}

operator fun Double.times(matrix: Matrix) = matrix * this
```

```kotlin
package algebra.real

data class Vector(
    private val elements: List<Double>,
) {

    // Providing a varargs constructor is an extension
    constructor(vararg elements: Double) : this(elements.toList())

    val length = elements.size

    init {
        if (length <= 0) throw IllegalArgumentException()
    }

    operator fun get(index: Int): Double =
        if (index !in 0..<length) throw IndexOutOfBoundsException() else elements[index]

    operator fun plus(other: Vector): Vector =
        if (length != other.length) {
            throw UnsupportedOperationException()
        } else {
            Vector(elements.zip(other.elements).map { (a, b) -> a + b })
        }

    operator fun times(scalar: Double): Vector =
        Vector(elements.map { it * scalar })

    infix fun dot(other: Vector): Double =
        if (length != other.length) {
            throw UnsupportedOperationException()
        } else {
            elements.zip(other.elements).map { (a, b) -> a * b }
                .reduce(Double::plus)
        }

    override fun toString(): String = elements.joinToString(prefix = "(", postfix = ")")

    // Providing iterability is an extension
    operator fun iterator(): Iterator<Double> = object : Iterator<Double> {
        private var index: Int = 0

        override fun hasNext(): Boolean = index < length

        override fun next(): Double = this@Vector[index++]
    }
}

operator fun Double.times(vector: Vector) = vector * this
```

```kotlin
package algebra.generic

import kotlin.test.Test
import kotlin.test.assertEquals
import kotlin.test.fail

class DoubleMatrixTests {

    private val factory = AlgebraFactory(Double::plus, Double::times)

    @Test
    fun ‘get row‘() {
        val m1 = factory.makeMatrix(
            listOf(
                listOf(1.0, 2.0, 3.0, 0.5, 1.0),
                listOf(0.0, 1.0, 0.0, 2.0, 3.0),
                listOf(1.0, 0.0, 1.0, 2.0, 4.0),
                listOf(2.0, 0.0, 1.0, 1.0, 1.0),
            ),
        )
        assertEquals(factory.makeVector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)), m1.getRow(0))
        assertEquals(factory.makeVector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)), m1.getRow(1))
        assertEquals(factory.makeVector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)), m1.getRow(2))
        assertEquals(factory.makeVector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)), m1.getRow(3))

        assertEquals(factory.makeVector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)), m1[0])
        assertEquals(factory.makeVector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)), m1[1])
        assertEquals(factory.makeVector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)), m1[2])
        assertEquals(factory.makeVector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)), m1[3])
    }

    @Test
    fun ‘get column‘() {
        val m1 = factory.makeMatrix(
            listOf(
                listOf(1.0, 2.0, 3.0, 0.5, 1.0),
                listOf(0.0, 1.0, 0.0, 2.0, 3.0),
                listOf(1.0, 0.0, 1.0, 2.0, 4.0),
                listOf(2.0, 0.0, 1.0, 1.0, 1.0),
            ),
        )
        assertEquals(factory.makeVector(listOf(1.0, 0.0, 1.0, 2.0)), m1.getColumn(0))
        assertEquals(factory.makeVector(listOf(2.0, 1.0, 0.0, 0.0)), m1.getColumn(1))
        assertEquals(factory.makeVector(listOf(3.0, 0.0, 1.0, 1.0)), m1.getColumn(2))
        assertEquals(factory.makeVector(listOf(0.5, 2.0, 2.0, 1.0)), m1.getColumn(3))
        assertEquals(factory.makeVector(listOf(1.0, 3.0, 4.0, 1.0)), m1.getColumn(4))
    }

    @Test
    fun ‘get element‘() {
        val m1 = factory.makeMatrix(
            listOf(
                listOf(1.0, 2.0, 3.0, 0.5, 1.0),
                listOf(0.0, 1.0, 0.0, 2.0, 3.0),
                listOf(1.0, 0.0, 1.0, 2.0, 4.0),
                listOf(2.0, 0.0, 1.0, 1.0, 1.0),
            ),
        )
        assertEquals(1.0, m1[0, 0])
```

```kotlin
        assertEquals(2.0, m1[0, 1])
        assertEquals(3.0, m1[0, 2])
        assertEquals(0.50, m1[0, 3])
        assertEquals(1.0, m1[0, 4])
        assertEquals(0.0, m1[1, 0])
        assertEquals(1.0, m1[1, 1])
        assertEquals(0.0, m1[1, 2])
        assertEquals(2.0, m1[1, 3])
        assertEquals(3.0, m1[1, 4])
        assertEquals(1.0, m1[2, 0])
        assertEquals(0.0, m1[2, 1])
        assertEquals(1.0, m1[2, 2])
        assertEquals(2.0, m1[2, 3])
        assertEquals(4.0, m1[2, 4])
        assertEquals(2.0, m1[3, 0])
        assertEquals(0.0, m1[3, 1])
        assertEquals(1.0, m1[3, 2])
        assertEquals(1.0, m1[3, 3])
        assertEquals(1.0, m1[3, 4])
    }

    @Test
    fun ‘matrix multiplication simple‘() {
        val m1 = factory.makeMatrix(
            listOf(
                listOf(1.0, 1.0),
            ),
        )
        val m2 = factory.makeMatrix(
            listOf(
                listOf(1.0),
                listOf(1.0),
            ),
        )
        val product = m1 * m2
        assertEquals(factory.makeMatrix(listOf(listOf(2.0))), product)
    }

    @Test
    fun ‘matrix multiplication‘() {
        val m1 = factory.makeMatrix(
            listOf(
                listOf(1.0, 2.0, 3.0, 0.5, 1.0),
                listOf(0.0, 1.0, 0.0, 2.0, 3.0),
                listOf(1.0, 0.0, 1.0, 2.0, 4.0),
                listOf(2.0, 0.0, 1.0, 1.0, 1.0),
            ),
        )

        val m2 = factory.makeMatrix(
            listOf(
                listOf(2.0, 3.0),
                listOf(1.0, 2.0),
                listOf(4.0, 1.0),
                listOf(0.0, 1.0),
                listOf(1.0, 3.0),
            ),
        )

        val product = factory.makeMatrix(
            listOf(
                listOf(17.0, 13.5),
                listOf(4.0, 13.0),
                listOf(10.0, 18.0),
                listOf(9.0, 11.0),
            ),
        )

        assertEquals(product, m1 * m2)
    }

    @Test
    fun ‘matrix addition‘() {
        val m1 = factory.makeMatrix(
            listOf(
                listOf(1.0, 2.0, 3.0, 0.5, 1.0),
                listOf(0.0, 1.0, 0.0, 2.0, 3.0),
                listOf(1.0, 0.0, 1.0, 2.0, 4.0),
                listOf(2.0, 0.0, 1.0, 1.0, 1.0),
            ),
        )

        val m2 = factory.makeMatrix(
            listOf(
                listOf(11.0, 12.0, 13.0, 10.5, 11.0),
                listOf(10.0, 11.0, 10.0, 12.0, 13.0),
                listOf(11.0, 10.0, 11.0, 12.0, 14.0),
                listOf(12.0, 10.0, 11.0, 11.0, 11.0),
            ),
        )

        val sum = factory.makeMatrix(
            listOf(
                listOf(12.0, 14.0, 16.0, 11.0, 12.0),
                listOf(10.0, 12.0, 10.0, 14.0, 16.0),
                listOf(12.0, 10.0, 12.0, 14.0, 18.0),
                listOf(14.0, 10.0, 12.0, 12.0, 12.0),
            ),
        )

        assertEquals(sum, m1 + m2)
    }

    @Test
    fun ‘left multiply by scalar‘() {
        val m1 = factory.makeMatrix(
            listOf(
                listOf(1.0, 2.0, 4.0, 0.5, 1.0),
                listOf(0.0, 1.0, 0.0, 2.0, 4.0),
                listOf(1.0, 0.0, 1.0, 2.0, 4.0),
                listOf(2.0, 0.0, 1.0, 1.0, 1.0),
            ),
        )

        val scaled = factory.makeMatrix(
            listOf(
                listOf(.10, .20, .40, .05, .10),
                listOf(.00, .10, .00, .20, .40),
                listOf(.10, .00, .10, .20, .40),
                listOf(.20, .00, .10, .10, .10),
            ),
        )

        assertEquals(scaled, 0.1 * m1)
    }

    @Test
    fun ‘right multiply by scalar‘() {
        val m1 = factory.makeMatrix(
            listOf(
                listOf(1.0, 2.0, 4.0, 0.5, 1.0),
                listOf(0.0, 1.0, 0.0, 2.0, 4.0),
                listOf(1.0, 0.0, 1.0, 2.0, 4.0),
                listOf(2.0, 0.0, 1.0, 1.0, 1.0),
            ),
        )
```

```kotlin

        val scaled = factory.makeMatrix(
            listOf(
                listOf(.10, .20, .40, .05, .10),
                listOf(.00, .10, .00, .20, .40),
                listOf(.10, .00, .10, .20, .40),
                listOf(.20, .00, .10, .10, .10),
            ),
        )

        assertEquals(scaled, m1 * 0.1)
    }

    @Test
    fun ‘string  representation‘() {
        val m1 = factory.makeMatrix(
            listOf(
                listOf(1.46, 2.0, 4.0, 0.5, 1.0),
                listOf(0.0, 1.0, 100.0, 2.0, 4.0),
                listOf(1.0, 0.0, 1.0, 2020.12, 4.0),
                listOf(2.0, 0.0, 1.0, 1.0, 1.0),
            ),
        )
        val stringRepresentation =
            """
                [ 1.46 2.0   4.0     0.5 1.0 ]
                [  0.0 1.0 100.0     2.0 4.0 ]
                [  1.0 0.0   1.0 2020.12 4.0 ]
                [  2.0 0.0   1.0     1.0 1.0 ]
            """.trimIndent()
        assertEquals(stringRepresentation, m1.toString())
    }

    @Test
    fun ‘exception - empty matrix‘() {
        try {
            factory.makeMatrix(emptyList())
            fail("IllegalArgumentException was expected.")
        } catch (exception: IllegalArgumentException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - negative row index‘() {
        try {
            factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0, 1.0))).getRow(-1)
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - negative row index with operator‘() {
        try {
            factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0, 1.0)))[-1]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - negative column index‘() {
        try {
            factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0,
1.0))).getColumn(-1)
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - too large row index‘() {
        try {
            factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0, 1.0))).getRow(2)
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - too large row index with operator‘() {
        try {
            factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0, 1.0)))[2]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - too large column index‘() {
        try {
            factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0,
1.0))).getColumn(2)
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - indices out of bounds‘() {
        val m = factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0, 1.0)))
        try {
            val entry = m[-1, 1]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
        try {
            val entry = m[1, -1]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
        try {
            val entry = m[2, 0]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
        try {
            val entry = m[0, 2]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
```

```kotlin
    fun ‘exception - add matrices with different row counts‘() {
        val m1 = factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0, 1.0)))
        val m2 = factory.makeMatrix(listOf(listOf(1.0, 1.0)))
        try {
            m1 + m2
            fail("UnsupportedOperationException was expected")
        } catch (exception: UnsupportedOperationException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - add matrices with different column counts‘() {
        val m1 = factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0, 1.0)))
        val m2 = factory.makeMatrix(listOf(listOf(1.0, 1.0, 1.0), listOf(1.0, 1.0,
1.0)))
        try {
            m1 + m2
            fail("UnsupportedOperationException was expected")
        } catch (exception: UnsupportedOperationException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - multiply matrices with incompatible sizes‘() {
        val m1 = factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0, 1.0)))
        val m2 = factory.makeMatrix(
            listOf(
                listOf(1.0, 1.0, 1.0),
                listOf(1.0, 1.0, 1.0),
                listOf(1.0, 1.0, 1.0),
            ),
        )
        try {
            m1 * m2
            fail("UnsupportedOperationException was expected")
        } catch (exception: UnsupportedOperationException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - create matrix with mismatched column counts‘() {
        try {
            factory.makeMatrix(
                listOf(
                    listOf(1.0, 1.0),
                    listOf(1.0, 1.0, 1.0),
                ),
            )
            fail("IllegalArgumentException was expected")
        } catch (exception: IllegalArgumentException) {
            // Good: exception was expected.
        }
    }
}
```

```kotlin
package algebra.generic

import kotlin.test.Test
import kotlin.test.assertEquals
import kotlin.test.fail

class DoubleVectorTests {

    private val factory = AlgebraFactory(Double::plus, Double::times)

    @Test
    fun ‘vector length‘() {
        val v = factory.makeVector(listOf(1.0, 2.0, 3.0))
        assertEquals(3, v.length)
    }

    @Test
    fun ‘get from vector‘() {
        val v = factory.makeVector(listOf(1.0, 2.0, 3.0))
        assertEquals(1.0, v[0])
        assertEquals(2.0, v[1])
        assertEquals(3.0, v[2])
    }

    @Test
    fun ‘vector addition‘() {
        val v1 = factory.makeVector(listOf(1.0, 2.0, 3.0))
        val v2 = factory.makeVector(listOf(4.0, 5.0, 6.0))
        val sum = factory.makeVector(listOf(5.0, 7.0, 9.0))
        assertEquals(sum, v1 + v2)
    }

    @Test
    fun ‘scalar times vector‘() {
        val v1 = factory.makeVector(listOf(1.0, 2.0, 3.0))
        val scaled = factory.makeVector(listOf(10.0, 20.0, 30.0))
        assertEquals(scaled, v1 * 10.0)
    }

    @Test
    fun ‘vector times scalar‘() {
        val v1 = factory.makeVector(listOf(1.0, 2.0, 3.0))
        val scaled = factory.makeVector(listOf(10.0, 20.0, 30.0))
        assertEquals(scaled, 10.0 * v1)
    }

    @Test
    fun ‘dot product‘() {
        val v1 = factory.makeVector(listOf(1.0, 0.0, 0.0))
        val v2 = factory.makeVector(listOf(0.0, 1.0, 0.0))
        assertEquals(0.0, v1 dot v2)
    }

    @Test
    fun ‘dot product larger vectors‘() {
        val v1 = factory.makeVector(listOf(1.0, 2.0, 3.0, 4.0, 5.0))
        val v2 = factory.makeVector(listOf(6.0, 7.0, 8.0, 9.0, 10.0))
        assertEquals(130.0, v1 dot v2)
    }

    @Test
    fun ‘string representation‘() {
        val v1 = factory.makeVector(listOf(1.0, 2.0, 3.0, 4.0, 5.0))
        assertEquals("(1.0, 2.0, 3.0, 4.0, 5.0)", v1.toString())
    }

    @Test
    fun ‘exception - empty vector‘() {
```

```kotlin
        try {
            factory.makeVector(emptyList())
            fail("IllegalArgumentException was expected.")
        } catch (exception: IllegalArgumentException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - lengths do not match in addition‘() {
        try {
            factory.makeVector(listOf(1.0, 2.0)) + factory.makeVector(listOf(1.0,
2.0, 3.0))
            fail("UnsupportedOperationException was expected.")
        } catch (exception: UnsupportedOperationException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - lengths do not match in dot product‘() {
        try {
            factory.makeVector(listOf(1.0, 2.0)) dot factory.makeVector(listOf(1.0,
2.0, 3.0))
            fail("UnsupportedOperationException was expected.")
        } catch (exception: UnsupportedOperationException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - get at negative index‘() {
        try {
            factory.makeVector(listOf(1.0, 2.0))[-1]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - get at too large index‘() {
        try {
            factory.makeVector(listOf(1.0, 2.0))[2]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }
}
```

```kotlin
package algebra.generic

import kotlin.test.Test
import kotlin.test.assertEquals

class MatrixExtensionTests {

    @Test
    fun `iterate over varargs-constructed matrix`() {
        val plus: (Int, Int) -> Int = { a, b -> a + b }
        val times: (Int, Int) -> Int = { a, b -> a + b }

        val m1 = Matrix<Int>(
            plus,
            times,
            Vector(plus, times, listOf(10, 20, 30, 5, 10)),
            Vector(plus, times, listOf(0, 10, 0, 20, 30)),
            Vector(plus, times, listOf(10, 0, 10, 20, 40)),
            Vector(plus, times, listOf(20, 0, 10, 10, 10)),
        )
        val rows = mutableListOf<Vector<Int>>()
        for (row in m1) {
            rows.add(row)
        }
        val expectedRows = listOf(
            Vector(plus, times, listOf(10, 20, 30, 5, 10)),
            Vector(plus, times, listOf(0, 10, 0, 20, 30)),
            Vector(plus, times, listOf(10, 0, 10, 20, 40)),
            Vector(plus, times, listOf(20, 0, 10, 10, 10)),
        )
        assertEquals(expectedRows, rows)
    }

    @Test
    fun `get element`() {
        val plus: (Int, Int) -> Int = { a, b -> a + b }
        val times: (Int, Int) -> Int = { a, b -> a + b }

        val m1 = Matrix(
            plus,
            times,
            Vector(plus, times, listOf(10, 20, 30, 5, 10)),
            Vector(plus, times, listOf(0, 10, 0, 20, 30)),
            Vector(plus, times, listOf(10, 0, 10, 20, 40)),
            Vector(plus, times, listOf(20, 0, 10, 10, 10)),
        )
        assertEquals(10, m1[0, 0])
        assertEquals(20, m1[0, 1])
        assertEquals(30, m1[0, 2])
        assertEquals(5, m1[0, 3])
        assertEquals(10, m1[0, 4])
        assertEquals(0, m1[1, 0])
        assertEquals(10, m1[1, 1])
        assertEquals(0, m1[1, 2])
        assertEquals(20, m1[1, 3])
        assertEquals(30, m1[1, 4])
        assertEquals(10, m1[2, 0])
        assertEquals(0, m1[2, 1])
        assertEquals(10, m1[2, 2])
        assertEquals(20, m1[2, 3])
        assertEquals(40, m1[2, 4])
        assertEquals(20, m1[3, 0])
        assertEquals(0, m1[3, 1])
        assertEquals(10, m1[3, 2])
        assertEquals(10, m1[3, 3])
        assertEquals(10, m1[3, 4])
    }
}
```

```kotlin
package algebra.generic

import kotlin.test.Test
import kotlin.test.assertEquals

class NestedMatrixTests {

    private val innerFactory = AlgebraFactory(Int::plus, Int::times)

    private val outerFactory = AlgebraFactory(Matrix<Int>::plus, Matrix<Int>::times)

    @Test
    fun `add nested matrices`() {
        val allZeroes2x2 = innerFactory.makeMatrix(
            listOf(
                listOf(0, 0),
                listOf(0, 0),
            ),
        )

        val allOnes2x2 = innerFactory.makeMatrix(
            listOf(
                listOf(1, 1),
                listOf(1, 1),
            ),
        )

        val m1 = outerFactory.makeMatrix(
            listOf(
                listOf(allZeroes2x2, allOnes2x2),
                listOf(allOnes2x2, allZeroes2x2),
            ),
        )
        val m2 = outerFactory.makeMatrix(
            listOf(
                listOf(allOnes2x2, allZeroes2x2),
                listOf(allZeroes2x2, allOnes2x2),
            ),
        )
        val expectedSum = outerFactory.makeMatrix(
            listOf(
                listOf(allOnes2x2, allOnes2x2),
                listOf(allOnes2x2, allOnes2x2),
            ),
        )
        assertEquals(expectedSum, m1 + m2)
    }

    private fun make2x2Diagonal(value: Int): Matrix<Int> =
        innerFactory.makeMatrix(
            listOf(
                listOf(value, 0),
                listOf(0, value),
            ),
        )

    @Test
    fun `multiply nested matrices`() {
        val intMatrix1 = innerFactory.makeMatrix(
            listOf(
                listOf(1, 2),
                listOf(3, 4),
            ),
        )

        val intMatrix2 = innerFactory.makeMatrix(
            listOf(
                listOf(5, 6),
                listOf(7, 8),
            ),
        )

        val intMatrixProduct1 = innerFactory.makeMatrix(
            listOf(
                listOf(19, 22),
                listOf(43, 50),
            ),
        )

        val intMatrixProduct2 = innerFactory.makeMatrix(
            listOf(
                listOf(23, 34),
                listOf(31, 46),
            ),
        )

        assertEquals(intMatrixProduct1, intMatrix1 * intMatrix2)

        assertEquals(intMatrixProduct2, intMatrix2 * intMatrix1)

        val nestedMatrix1 = outerFactory.makeMatrix(
            listOf(
                listOf(make2x2Diagonal(1), make2x2Diagonal(2)),
                listOf(make2x2Diagonal(3), make2x2Diagonal(4)),
            ),
        )

        val nestedMatrix2 = outerFactory.makeMatrix(
            listOf(
                listOf(make2x2Diagonal(5), make2x2Diagonal(6)),
                listOf(make2x2Diagonal(7), make2x2Diagonal(8)),
            ),
        )

        val nestedMatrixProduct1 = outerFactory.makeMatrix(
            listOf(
                listOf(make2x2Diagonal(19), make2x2Diagonal(22)),
                listOf(make2x2Diagonal(43), make2x2Diagonal(50)),
            ),
        )

        val nestedMatrixProduct2 = outerFactory.makeMatrix(
            listOf(
                listOf(make2x2Diagonal(23), make2x2Diagonal(34)),
                listOf(make2x2Diagonal(31), make2x2Diagonal(46)),
            ),
        )

        assertEquals(nestedMatrixProduct1, nestedMatrix1 * nestedMatrix2)

        assertEquals(nestedMatrixProduct2, nestedMatrix2 * nestedMatrix1)
    }

    @Test
    fun `left and right multiplication by matrix scalar`() {
        val m1 = innerFactory.makeMatrix(
            listOf(
                listOf(1, 2),
                listOf(1, 2),
            ),
        )

        val m2 = innerFactory.makeMatrix(
            listOf(
                listOf(1, 3),
                listOf(1, 3),
```

```
            ),
        )

        val m1xm1 = innerFactory.makeMatrix(
            listOf(
                listOf(3, 6),
                listOf(3, 6),
            ),
        )

        val m2xm2 = innerFactory.makeMatrix(
            listOf(
                listOf(4, 12),
                listOf(4, 12),
            ),
        )

        val m1xm2 = innerFactory.makeMatrix(
            listOf(
                listOf(3, 9),
                listOf(3, 9),
            ),
        )

        val m2xm1 = innerFactory.makeMatrix(
            listOf(
                listOf(4, 8),
                listOf(4, 8),
            ),
        )

        assertEquals(m1xm1, m1 * m1)
        assertEquals(m2xm2, m2 * m2)
        assertEquals(m1xm2, m1 * m2)
        assertEquals(m2xm1, m2 * m1)

        val nestedMatrix = outerFactory.makeMatrix(
            listOf(
                listOf(m1, m2, m1, m2),
                listOf(m2, m1, m2, m1),
            ),
        )

        val nestedMatrixLeftScaledByM1 = outerFactory.makeMatrix(
            listOf(
                listOf(m1xm1, m1xm2, m1xm1, m1xm2),
                listOf(m1xm2, m1xm1, m1xm2, m1xm1),
            ),
        )

        val nestedMatrixRightScaledByM1 = outerFactory.makeMatrix(
            listOf(
                listOf(m1xm1, m2xm1, m1xm1, m2xm1),
                listOf(m2xm1, m1xm1, m2xm1, m1xm1),
            ),
        )

        val nestedMatrixLeftScaledByM2 = outerFactory.makeMatrix(
            listOf(
                listOf(m2xm1, m2xm2, m2xm1, m2xm2),
                listOf(m2xm2, m2xm1, m2xm2, m2xm1),
            ),
        )

        val nestedMatrixRightScaledByM2 = outerFactory.makeMatrix(
            listOf(
                listOf(m1xm2, m2xm2, m1xm2, m2xm2),
                listOf(m2xm2, m1xm2, m2xm2, m1xm2),
```

```kotlin
            ),
        )

        assertEquals(nestedMatrixLeftScaledByM1, m1 * nestedMatrix)
        assertEquals(nestedMatrixRightScaledByM1, nestedMatrix * m1)
        assertEquals(nestedMatrixLeftScaledByM2, m2 * nestedMatrix)
        assertEquals(nestedMatrixRightScaledByM2, nestedMatrix * m2)
    }
}
```

```kotlin
package algebra.generic

import org.junit.Test
import kotlin.test.assertEquals

class StringMatrixes {

    @Test
    fun ‘mutlitply string matrices‘() {
        val stringAlgebraFactory = AlgebraFactory<String>(
            plus = { a, b -> a + "+" + b },
            times = { a, b -> a + "*" + b },
        )

        val m1 = stringAlgebraFactory.makeMatrix(
            listOf(
                listOf("ant", "bug", "croc"),
                listOf("deer", "elephant", "frog"),
            ),
        )

        val m2 = stringAlgebraFactory.makeMatrix(
            listOf(
                listOf("wasp", "beetle"),
                listOf("goblin", "midge"),
                listOf("mite", "kangaroo"),
            ),
        )

        val product = stringAlgebraFactory.makeMatrix(
            listOf(
                listOf("ant*wasp+bug*goblin+croc*mite", "ant*beetle+bug*midge+croc*kangaroo"),
                listOf("deer*wasp+elephant*goblin+frog*mite", "deer*beetle+elephant*midge+frog*kangaroo"),
            ),
        )

        assertEquals(product, m1 * m2)

        val expectedString =
            """
                [      ant*wasp+bug*goblin+croc*mite ant*beetle+bug*midge+croc*kangaroo ]
                [ deer*wasp+elephant*goblin+frog*mite deer*beetle+elephant*midge+frog*kangaroo ]
            """.trimIndent()

        assertEquals(expectedString, product.toString())
    }
}
```

```kotlin
package algebra.generic

import kotlin.test.Test
import kotlin.test.assertEquals

class VectorExtensionTests {

    @Test
    fun `iterate over varargs-constructed vector`() {
        val v = Vector({ s, t -> "$s+$t" }, { s, t -> "$s*$t" }, "a", "b", "c", 
"d", "e")
        val elements = mutableListOf<String>()
        for (element in v) {
            elements.add(element)
        }
        assertEquals(listOf("a", "b", "c", "d", "e"), elements)
    }

    @Test
    fun `vector length`() {
        val v = Vector({ s, t -> "$s+$t" }, { s, t -> "$s*$t" }, "a", "b", "c", 
"d", "e")
        assertEquals(5, v.length)
    }

    @Test
    fun `get from vector`() {
        val v = Vector({ s, t -> "$s+$t" }, { s, t -> "$s*$t" }, "a", "b", "c", 
"d", "e")
        assertEquals("a", v[0])
        assertEquals("b", v[1])
        assertEquals("c", v[2])
        assertEquals("d", v[3])
        assertEquals("e", v[4])
    }
}
```

```kotlin
package algebra.real

import kotlin.test.Test
import kotlin.test.assertEquals

class MatrixExtensionTests {

    @Test
    fun `iterate over varargs-constructed matrix`() {
        val m1 = Matrix(
            Vector(1.0, 2.0, 3.0, 0.5, 1.0),
            Vector(0.0, 1.0, 0.0, 2.0, 3.0),
            Vector(1.0, 0.0, 1.0, 2.0, 4.0),
            Vector(2.0, 0.0, 1.0, 1.0, 1.0),
        )
        val rows = mutableListOf<Vector>()
        for (row in m1) {
            rows.add(row)
        }
        val expectedRows = listOf(
            Vector(1.0, 2.0, 3.0, 0.5, 1.0),
            Vector(0.0, 1.0, 0.0, 2.0, 3.0),
            Vector(1.0, 0.0, 1.0, 2.0, 4.0),
            Vector(2.0, 0.0, 1.0, 1.0, 1.0),
        )
        assertEquals(expectedRows, rows)
    }

    @Test
    fun `get element`() {
        val m1 = Matrix(
            Vector(1.0, 2.0, 3.0, 0.5, 1.0),
            Vector(0.0, 1.0, 0.0, 2.0, 3.0),
            Vector(1.0, 0.0, 1.0, 2.0, 4.0),
            Vector(2.0, 0.0, 1.0, 1.0, 1.0),
        )
        assertEquals(1.0, m1[0, 0])
        assertEquals(2.0, m1[0, 1])
        assertEquals(3.0, m1[0, 2])
        assertEquals(0.50, m1[0, 3])
        assertEquals(1.0, m1[0, 4])
        assertEquals(0.0, m1[1, 0])
        assertEquals(1.0, m1[1, 1])
        assertEquals(0.0, m1[1, 2])
        assertEquals(2.0, m1[1, 3])
        assertEquals(3.0, m1[1, 4])
        assertEquals(1.0, m1[2, 0])
        assertEquals(0.0, m1[2, 1])
        assertEquals(1.0, m1[2, 2])
        assertEquals(2.0, m1[2, 3])
        assertEquals(4.0, m1[2, 4])
        assertEquals(2.0, m1[3, 0])
        assertEquals(0.0, m1[3, 1])
        assertEquals(1.0, m1[3, 2])
        assertEquals(1.0, m1[3, 3])
        assertEquals(1.0, m1[3, 4])
    }
}
```

```kotlin
package algebra.real

import kotlin.test.Test
import kotlin.test.assertEquals
import kotlin.test.fail

class MatrixTests {

    @Test
    fun `get row`() {
        val m1 = Matrix(
            listOf(
                Vector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)),
                Vector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)),
                Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)),
                Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
            ),
        )
        assertEquals(Vector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)), m1.getRow(0))
        assertEquals(Vector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)), m1.getRow(1))
        assertEquals(Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)), m1.getRow(2))
        assertEquals(Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)), m1.getRow(3))

        assertEquals(Vector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)), m1[0])
        assertEquals(Vector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)), m1[1])
        assertEquals(Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)), m1[2])
        assertEquals(Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)), m1[3])
    }

    @Test
    fun `get column`() {
        val m1 = Matrix(
            listOf(
                Vector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)),
                Vector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)),
                Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)),
                Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
            ),
        )
        assertEquals(Vector(listOf(1.0, 0.0, 1.0, 2.0)), m1.getColumn(0))
        assertEquals(Vector(listOf(2.0, 1.0, 0.0, 0.0)), m1.getColumn(1))
        assertEquals(Vector(listOf(3.0, 0.0, 1.0, 1.0)), m1.getColumn(2))
        assertEquals(Vector(listOf(0.5, 2.0, 2.0, 1.0)), m1.getColumn(3))
        assertEquals(Vector(listOf(1.0, 3.0, 4.0, 1.0)), m1.getColumn(4))
    }

    @Test
    fun `get element`() {
        val m1 = Matrix(
            listOf(
                Vector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)),
                Vector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)),
                Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)),
                Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
            ),
        )
        assertEquals(1.0, m1[0, 0])
        assertEquals(2.0, m1[0, 1])
        assertEquals(3.0, m1[0, 2])
        assertEquals(0.50, m1[0, 3])
        assertEquals(1.0, m1[0, 4])
        assertEquals(0.0, m1[1, 0])
        assertEquals(1.0, m1[1, 1])
        assertEquals(0.0, m1[1, 2])
        assertEquals(2.0, m1[1, 3])
        assertEquals(3.0, m1[1, 4])
        assertEquals(1.0, m1[2, 0])
        assertEquals(0.0, m1[2, 1])
```

```kotlin
        assertEquals(1.0, m1[2, 2])
        assertEquals(2.0, m1[2, 3])
        assertEquals(4.0, m1[2, 4])
        assertEquals(2.0, m1[3, 0])
        assertEquals(0.0, m1[3, 1])
        assertEquals(1.0, m1[3, 2])
        assertEquals(1.0, m1[3, 3])
        assertEquals(1.0, m1[3, 4])
    }

    @Test
    fun `matrix multiplication simple`() {
        val m1 = Matrix(
            listOf(
                Vector(listOf(1.0, 1.0)),
            ),
        )
        val m2 = Matrix(
            listOf(
                Vector(listOf(1.0)),
                Vector(listOf(1.0)),
            ),
        )
        val product = m1 * m2
        assertEquals(Matrix(listOf(Vector(listOf(2.0)))), product)
    }

    @Test
    fun `matrix multiplication`() {
        val m1 = Matrix(
            listOf(
                Vector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)),
                Vector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)),
                Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)),
                Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
            ),
        )

        val m2 = Matrix(
            listOf(
                Vector(listOf(2.0, 3.0)),
                Vector(listOf(1.0, 2.0)),
                Vector(listOf(4.0, 1.0)),
                Vector(listOf(0.0, 1.0)),
                Vector(listOf(1.0, 3.0)),
            ),
        )

        val product = Matrix(
            listOf(
                Vector(listOf(17.0, 13.5)),
                Vector(listOf(4.0, 13.0)),
                Vector(listOf(10.0, 18.0)),
                Vector(listOf(9.0, 11.0)),
            ),
        )

        assertEquals(product, m1 * m2)
    }

    @Test
    fun `matrix addition`() {
        val m1 = Matrix(
            listOf(
                Vector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)),
                Vector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)),
                Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)),
                Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
            ),
        )

        val m2 = Matrix(
            listOf(
                Vector(listOf(11.0, 12.0, 13.0, 10.5, 11.0)),
                Vector(listOf(10.0, 11.0, 10.0, 12.0, 13.0)),
                Vector(listOf(11.0, 10.0, 11.0, 12.0, 14.0)),
                Vector(listOf(12.0, 10.0, 11.0, 11.0, 11.0)),
            ),
        )

        val sum = Matrix(
            listOf(
                Vector(listOf(12.0, 14.0, 16.0, 11.0, 12.0)),
                Vector(listOf(10.0, 12.0, 10.0, 14.0, 16.0)),
                Vector(listOf(12.0, 10.0, 12.0, 14.0, 18.0)),
                Vector(listOf(14.0, 10.0, 12.0, 12.0, 12.0)),
            ),
        )

        assertEquals(sum, m1 + m2)
    }

    @Test
    fun ‘left multiply by scalar‘() {
        val m1 = Matrix(
            listOf(
                Vector(listOf(1.0, 2.0, 4.0, 0.5, 1.0)),
                Vector(listOf(0.0, 1.0, 0.0, 2.0, 4.0)),
                Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)),
                Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
            ),
        )

        val scaled = Matrix(
            listOf(
                Vector(listOf(.10, .20, .40, .05, .10)),
                Vector(listOf(.00, .10, .00, .20, .40)),
                Vector(listOf(.10, .00, .10, .20, .40)),
                Vector(listOf(.20, .00, .10, .10, .10)),
            ),
        )

        assertEquals(scaled, 0.1 * m1)
    }

    @Test
    fun ‘right multiply by scalar‘() {
        val m1 = Matrix(
            listOf(
                Vector(listOf(1.0, 2.0, 4.0, 0.5, 1.0)),
                Vector(listOf(0.0, 1.0, 0.0, 2.0, 4.0)),
                Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)),
                Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
            ),
        )

        val scaled = Matrix(
            listOf(
                Vector(listOf(.10, .20, .40, .05, .10)),
                Vector(listOf(.00, .10, .00, .20, .40)),
                Vector(listOf(.10, .00, .10, .20, .40)),
                Vector(listOf(.20, .00, .10, .10, .10)),
            ),
        )

        assertEquals(scaled, m1 * 0.1)
```

```kotlin
    }

    @Test
    fun ‘string  representation‘() {
        val m1 = Matrix(
            listOf(
                Vector(listOf(1.46, 2.0, 4.0, 0.5, 1.0)),
                Vector(listOf(0.0, 1.0, 100.0, 2.0, 4.0)),
                Vector(listOf(1.0, 0.0, 1.0, 2020.12, 4.0)),
                Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
            ),
        )
        val stringRepresentation =
            """
                [ 1.46 2.0   4.0     0.5 1.0 ]
                [  0.0 1.0 100.0     2.0 4.0 ]
                [  1.0 0.0   1.0 2020.12 4.0 ]
                [  2.0 0.0   1.0     1.0 1.0 ]
            """.trimIndent()

        assertEquals(stringRepresentation, m1.toString())
    }

    @Test
    fun ‘exception - empty matrix‘() {
        try {
            Matrix(emptyList())
            fail("IllegalArgumentException was expected.")
        } catch (exception: IllegalArgumentException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - negative row index‘() {
        try {
            Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0)))).getRow(-1)
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - negative row index with operator‘() {
        try {
            Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0))))[-1]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - negative column index‘() {
        try {
            Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0)))).getColumn(-1)
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - too large row index‘() {
        try {
            Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0)))).getRow(2)
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - too large row index with operator‘() {
        try {
            Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0))))[2]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - too large column index‘() {
        try {
            Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0)))).getColumn(2)
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - indices out of bounds‘() {
        val m = Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0))))
        try {
            val entry = m[-1, 1]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
        try {
            val entry = m[1, -1]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
        try {
            val entry = m[2, 0]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
        try {
            val entry = m[0, 2]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - add matrices with different row counts‘() {
        val m1 = Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0))))
        val m2 = Matrix(listOf(Vector(listOf(1.0, 1.0))))
        try {
            m1 + m2
            fail("UnsupportedOperationException was expected")
        } catch (exception: UnsupportedOperationException) {
            // Good: exception was expected.
        }
```

```
    }

    @Test
    fun ‘exception - add matrices with different column counts‘() {
        val m1 = Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0))))
        val m2 = Matrix(listOf(Vector(listOf(1.0, 1.0, 1.0)), Vector(listOf(1.0, 1.0, 1.0))))
        try {
            m1 + m2
            fail("UnsupportedOperationException was expected")
        } catch (exception: UnsupportedOperationException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - multiply matrices with incompatible sizes‘() {
        val m1 = Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0))))
        val m2 = Matrix(
            listOf(
                Vector(listOf(1.0, 1.0, 1.0)),
                Vector(listOf(1.0, 1.0, 1.0)),
                Vector(listOf(1.0, 1.0, 1.0)),
            ),
        )
        try {
            m1 * m2
            fail("UnsupportedOperationException was expected")
        } catch (exception: UnsupportedOperationException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - create matrix with mismatched column counts‘() {
        try {
            Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0, 1.0))))
            fail("IllegalArgumentException was expected")
        } catch (exception: IllegalArgumentException) {
            // Good: exception was expected.
        }
    }
}
```

```kotlin
package algebra.real

import kotlin.test.Test
import kotlin.test.assertEquals

class VectorExtensionTests {

    @Test
    fun ‘iterate over varargs-constructed vector‘() {
        val v1 = Vector(1.0, 2.0, 3.0, 4.0, 5.0)
        val elements = mutableListOf<Double>()
        for (element in v1) {
            elements.add(element)
        }
        assertEquals(listOf(1.0, 2.0, 3.0, 4.0, 5.0), elements)
    }

    @Test
    fun ‘vector length‘() {
        val v = Vector(1.0, 2.0, 3.0)
        assertEquals(3, v.length)
    }

    @Test
    fun ‘get from vector‘() {
        val v = Vector(1.0, 2.0, 3.0)
        assertEquals(1.0, v[0])
        assertEquals(2.0, v[1])
        assertEquals(3.0, v[2])
    }
}
```

```kotlin
package algebra.real

import kotlin.test.Test
import kotlin.test.assertEquals
import kotlin.test.fail

class VectorTests {

    @Test
    fun ‘vector length‘() {
        val v = Vector(listOf(1.0, 2.0, 3.0))
        assertEquals(3, v.length)
    }

    @Test
    fun ‘get from vector‘() {
        val v = Vector(listOf(1.0, 2.0, 3.0))
        assertEquals(1.0, v[0])
        assertEquals(2.0, v[1])
        assertEquals(3.0, v[2])
    }

    @Test
    fun ‘vector addition‘() {
        val v1 = Vector(listOf(1.0, 2.0, 3.0))
        val v2 = Vector(listOf(4.0, 5.0, 6.0))
        val sum = Vector(listOf(5.0, 7.0, 9.0))
        assertEquals(sum, v1 + v2)
    }

    @Test
    fun ‘scalar times vector‘() {
        val v1 = Vector(listOf(1.0, 2.0, 3.0))
        val scaled = Vector(listOf(10.0, 20.0, 30.0))
        assertEquals(scaled, v1 * 10.0)
    }

    @Test
    fun ‘vector times scalar‘() {
        val v1 = Vector(listOf(1.0, 2.0, 3.0))
        val scaled = Vector(listOf(10.0, 20.0, 30.0))
        assertEquals(scaled, 10.0 * v1)
    }

    @Test
    fun ‘dot product‘() {
        val v1 = Vector(listOf(1.0, 0.0, 0.0))
        val v2 = Vector(listOf(0.0, 1.0, 0.0))
        assertEquals(0.0, v1 dot v2)
    }

    @Test
    fun ‘dot product larger vectors‘() {
        val v1 = Vector(listOf(1.0, 2.0, 3.0, 4.0, 5.0))
        val v2 = Vector(listOf(6.0, 7.0, 8.0, 9.0, 10.0))
        assertEquals(130.0, v1 dot v2)
    }

    @Test
    fun ‘string representation‘() {
        val v1 = Vector(listOf(1.0, 2.0, 3.0, 4.0, 5.0))
        assertEquals("(1.0, 2.0, 3.0, 4.0, 5.0)", v1.toString())
    }

    @Test
    fun ‘exception - empty vector‘() {
        try {
            Vector(emptyList())
            fail("IllegalArgumentException was expected.")
        } catch (exception: IllegalArgumentException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - lengths do not match in addition‘() {
        try {
            Vector(listOf(1.0, 2.0)) + Vector(listOf(1.0, 2.0, 3.0))
            fail("UnsupportedOperationException was expected.")
        } catch (exception: UnsupportedOperationException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - lengths do not match in dot product‘() {
        try {
            Vector(listOf(1.0, 2.0)) dot Vector(listOf(1.0, 2.0, 3.0))
            fail("UnsupportedOperationException was expected.")
        } catch (exception: UnsupportedOperationException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - get at negative index‘() {
        try {
            Vector(listOf(1.0, 2.0))[-1]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }

    @Test
    fun ‘exception - get at too large index‘() {
        try {
            Vector(listOf(1.0, 2.0))[2]
            fail("IndexOutOfBoundsException was expected.")
        } catch (exception: IndexOutOfBoundsException) {
            // Good: exception was expected.
        }
    }
}
```