

```
1: Algebra
2: =====
3:
4: Aims:
5:
6: * Give the students experience working with data classes
7:
8: * Give the students experience working with operator overloading and
9:   infix functions
10:
11: * Give the students experience working with generic classes and
12:   function objects
13:
14: * Encourage good testing practices by having a code coverage target
15:   for tests
16:
17: Guide to breakdown of marks (out of 10):
18:
19: Do not give a student an A* if they achieve less than 90% test
20: coverage for any of their classes. In the PPT session for this lab,
21: emphasise the importance of thorough testing.
22:
23: Also do not give a student an A* if their solution features loops
24: (loops in their tests are fine). This is not in order to give them the
25: impression that loops are bad. Instead, it is to encourage them to
26: further explore the functional programming features that Kotlin
27: offers. Discussing the pros and cons of loops vs. a functional
28: approach will be a good point for discussing during the PPT.
29:
30: - 2 marks for implementing vectors over doubles
31:
32: - 3 marks for implementing matrices over doubles
33:
34: - 3 marks for lifting these vector and matrix implementations to a
35:   generic form
36:
37: - 2 marks for the quality of the new tests that are written
38:
39: Apart from the notes on when not to give an A* (which you should
40: enforce), this is just a guide - please use your judgement when
41: deciding how to score the exercise.
```

```
1: package algebra.generic
2:
3: class AlgebraFactory<T>{
4:     val plus: (T, T) -> T,
5:     val times: (T, T) -> T,
6: } {
7:     fun makeVector(elements: List<T>): Vector<T> = Vector(plus, times, elements)
8:
9:     fun makeMatrix(rows: List<List<T>>): Matrix<T> = Matrix(plus, times, rows.map {
it -> Vector(plus, times, it) })
10: }
```

```

1: package algebra.generic
2:
3: import algebra.real.times
4: import kotlin.IllegalArgumentException
5: import kotlin.math.max
6:
7: data class Matrix<T>{
8:     private val plus: (T, T) -> T,
9:     private val times: (T, T) -> T,
10:    private val rows: List<Vector<T>>,
11: } {
12:
13:     constructor(
14:         plus: (T, T) -> T,
15:         times: (T, T) -> T,
16:         vararg rows: Vector<T>,
17:     ) : this(plus, times, rows.toList())
18:
19:     val numRows = if (rows.isEmpty()) {
20:         throw IllegalArgumentException()
21:     } else {
22:         rows.size
23:     }
24:     val numColumns = rows[0].length
25:
26:     init {
27:         if (rows.any { it.length != numColumns }) {
28:             throw IllegalArgumentException()
29:         }
30:     }
31:
32:     operator fun get(rowIndex: Int, columnIndex: Int): T =
33:         if (rowIndex !in 0..

```

```

68:         throw UnsupportedOperationException()
69:     } else {
70:         copy(
71:             rows = rows.map { rowVector ->
72:                 Vector(
73:                     plus,
74:                     times,
75:                     (0..

```

```

1: package algebra.generic
2:
3: data class Vector<T>{
4:     private val plus: (T, T) -> T,
5:     private val times: (T, T) -> T,
6:     private val elements: List<T>,
7: } {
8:
9:     // Providing a varargs constructor is an extension
10:    constructor(
11:        plus: (T, T) -> T,
12:        times: (T, T) -> T,
13:        vararg elements: T,
14:    ) : this(plus, times, elements.toList())
15:
16:    val length = elements.size
17:
18:    init {
19:        if (length <= 0) throw IllegalArgumentException()
20:    }
21:
22:    operator fun T.plus(other: T): T = plus(this, other)
23:
24:    operator fun T.times(other: T): T = times(this, other)
25:
26:    operator fun get(index: Int): T =
27:        if (index !in 0..

```

```

1: package algebra.real
2:
3: import kotlin.math.max
4:
5: data class Matrix(private val rows: List<Vector>) {
6:
7:     constructor(vararg elements: Vector) : this(elements.toList())
8:
9:     val numRows = if (rows.isEmpty()) throw IllegalArgumentException() else
10:    rows.size
11:
12:     val numColumns = rows[0].length
13:
14:     init {
15:         if (rows.any { it.length != numColumns }) {
16:             throw IllegalArgumentException()
17:         }
18:     }
19:
20:     operator fun get(rowIndex: Int, columnIndex: Int): Double =
21:         if (rowIndex !in 0..

```

```

../solution/src/main/kotlin/algebra/real/Matrix.kt      Fri Jan 26 11:09:32 2024      2
68:      Matrix(rows.map { row -> scalar * row })
69:
70:      operator fun iterator(): Iterator<Vector> = object : Iterator<Vector> {
71:          private var index: Int = 0
72:
73:          override fun hasNext(): Boolean = index < numRows
74:
75:          override fun next(): Vector = this@Matrix[index++]
76:      }
77:
78:      override fun toString(): String {
79:          val largestColumnEntry: List<Int> = (0..

```

```

../solution/src/main/kotlin/algebra/real/Vector.kt      Fri Jan 26 11:09:32 2024      1
1:  package algebra.real
2:
3:  data class Vector(
4:      private val elements: List<Double>,
5:  ) {
6:
7:      // Providing a varargs constructor is an extension
8:      constructor(vararg elements: Double) : this(elements.toList())
9:
10:     val length = elements.size
11:
12:     init {
13:         if (length <= 0) throw IllegalArgumentException()
14:     }
15:
16:     operator fun get(index: Int): Double =
17:         if (index !in 0..

```

```

1: package algebra.generic
2:
3: import kotlin.test.Test
4: import kotlin.test.assertEquals
5: import kotlin.test.fail
6:
7: class DoubleMatrixTests {
8:
9:     private val factory = AlgebraFactory(Double::plus, Double::times)
10:
11:     @Test
12:     fun 'get row'() {
13:         val m1 = factory.makeMatrix(
14:             listOf(
15:                 listOf(1.0, 2.0, 3.0, 0.5, 1.0),
16:                 listOf(0.0, 1.0, 0.0, 2.0, 3.0),
17:                 listOf(1.0, 0.0, 1.0, 2.0, 4.0),
18:                 listOf(2.0, 0.0, 1.0, 1.0, 1.0),
19:             ),
20:         )
21:         assertEquals(factory.makeVector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)),
22:             m1.getRow(0))
23:         assertEquals(factory.makeVector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)),
24:             m1.getRow(1))
25:         assertEquals(factory.makeVector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)),
26:             m1.getRow(2))
27:         assertEquals(factory.makeVector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
28:             m1.getRow(3))
29:         assertEquals(factory.makeVector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)), m1[0])
30:         assertEquals(factory.makeVector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)), m1[1])
31:         assertEquals(factory.makeVector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)), m1[2])
32:         assertEquals(factory.makeVector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)), m1[3])
33:     }
34:
35:     @Test
36:     fun 'get column'() {
37:         val m1 = factory.makeMatrix(
38:             listOf(
39:                 listOf(1.0, 2.0, 3.0, 0.5, 1.0),
40:                 listOf(0.0, 1.0, 0.0, 2.0, 3.0),
41:                 listOf(1.0, 0.0, 1.0, 2.0, 4.0),
42:                 listOf(2.0, 0.0, 1.0, 1.0, 1.0),
43:             ),
44:         )
45:         assertEquals(factory.makeVector(listOf(1.0, 0.0, 1.0, 2.0)),
46:             m1.getColumn(0))
47:         assertEquals(factory.makeVector(listOf(2.0, 1.0, 0.0, 0.0)),
48:             m1.getColumn(1))
49:         assertEquals(factory.makeVector(listOf(3.0, 0.0, 1.0, 1.0)),
50:             m1.getColumn(2))
51:         assertEquals(factory.makeVector(listOf(0.5, 2.0, 2.0, 1.0)),
52:             m1.getColumn(3))
53:         assertEquals(factory.makeVector(listOf(1.0, 3.0, 4.0, 1.0)),
54:             m1.getColumn(4))
55:     }
56:
57:     @Test
58:     fun 'get element'() {
59:         val m1 = factory.makeMatrix(
60:             listOf(
61:                 listOf(1.0, 2.0, 3.0, 0.5, 1.0),
62:                 listOf(0.0, 1.0, 0.0, 2.0, 3.0),
63:                 listOf(1.0, 0.0, 1.0, 2.0, 4.0),
64:                 listOf(2.0, 0.0, 1.0, 1.0, 1.0),
65:             ),
66:         )
67:         assertEquals(1.0, m1[0, 0])

```

```

60:         assertEquals(2.0, m1[0, 1])
61:         assertEquals(3.0, m1[0, 2])
62:         assertEquals(0.5, m1[0, 3])
63:         assertEquals(1.0, m1[0, 4])
64:         assertEquals(0.0, m1[1, 0])
65:         assertEquals(1.0, m1[1, 1])
66:         assertEquals(0.0, m1[1, 2])
67:         assertEquals(2.0, m1[1, 3])
68:         assertEquals(3.0, m1[1, 4])
69:         assertEquals(1.0, m1[2, 0])
70:         assertEquals(0.0, m1[2, 1])
71:         assertEquals(1.0, m1[2, 2])
72:         assertEquals(2.0, m1[2, 3])
73:         assertEquals(4.0, m1[2, 4])
74:         assertEquals(2.0, m1[3, 0])
75:         assertEquals(0.0, m1[3, 1])
76:         assertEquals(1.0, m1[3, 2])
77:         assertEquals(1.0, m1[3, 3])
78:         assertEquals(1.0, m1[3, 4])
79:     }
80:
81:     @Test
82:     fun 'matrix multiplication simple'() {
83:         val m1 = factory.makeMatrix(
84:             listOf(
85:                 listOf(1.0, 1.0),
86:             ),
87:         )
88:         val m2 = factory.makeMatrix(
89:             listOf(
90:                 listOf(1.0),
91:                 listOf(1.0),
92:             ),
93:         )
94:         val product = m1 * m2
95:         assertEquals(factory.makeMatrix(listOf(listOf(2.0))), product)
96:     }
97:
98:     @Test
99:     fun 'matrix multiplication'() {
100:         val m1 = factory.makeMatrix(
101:             listOf(
102:                 listOf(1.0, 2.0, 3.0, 0.5, 1.0),
103:                 listOf(0.0, 1.0, 0.0, 2.0, 3.0),
104:                 listOf(1.0, 0.0, 1.0, 2.0, 4.0),
105:                 listOf(2.0, 0.0, 1.0, 1.0, 1.0),
106:             ),
107:         )
108:
109:         val m2 = factory.makeMatrix(
110:             listOf(
111:                 listOf(2.0, 3.0),
112:                 listOf(1.0, 2.0),
113:                 listOf(4.0, 1.0),
114:                 listOf(0.0, 1.0),
115:                 listOf(1.0, 3.0),
116:             ),
117:         )
118:
119:         val product = factory.makeMatrix(
120:             listOf(
121:                 listOf(17.0, 13.5),
122:                 listOf(4.0, 13.0),
123:                 listOf(10.0, 18.0),
124:                 listOf(9.0, 11.0),
125:             ),
126:         )
127:

```

```

128:     assertEquals(product, m1 * m2)
129: }
130:
131: @Test
132: fun 'matrix addition'() {
133:     val m1 = factory.makeMatrix(
134:         listOf(
135:             listOf(1.0, 2.0, 3.0, 0.5, 1.0),
136:             listOf(0.0, 1.0, 0.0, 2.0, 3.0),
137:             listOf(1.0, 0.0, 1.0, 2.0, 4.0),
138:             listOf(2.0, 0.0, 1.0, 1.0, 1.0),
139:         ),
140:     )
141:
142:     val m2 = factory.makeMatrix(
143:         listOf(
144:             listOf(11.0, 12.0, 13.0, 10.5, 11.0),
145:             listOf(10.0, 11.0, 10.0, 12.0, 13.0),
146:             listOf(11.0, 10.0, 11.0, 12.0, 14.0),
147:             listOf(12.0, 10.0, 11.0, 11.0, 11.0),
148:         ),
149:     )
150:
151:     val sum = factory.makeMatrix(
152:         listOf(
153:             listOf(12.0, 14.0, 16.0, 11.0, 12.0),
154:             listOf(10.0, 12.0, 10.0, 14.0, 16.0),
155:             listOf(12.0, 10.0, 12.0, 14.0, 18.0),
156:             listOf(14.0, 10.0, 12.0, 12.0, 12.0),
157:         ),
158:     )
159:
160:     assertEquals(sum, m1 + m2)
161: }
162:
163: @Test
164: fun 'left multiply by scalar'() {
165:     val m1 = factory.makeMatrix(
166:         listOf(
167:             listOf(1.0, 2.0, 4.0, 0.5, 1.0),
168:             listOf(0.0, 1.0, 0.0, 2.0, 4.0),
169:             listOf(1.0, 0.0, 1.0, 2.0, 4.0),
170:             listOf(2.0, 0.0, 1.0, 1.0, 1.0),
171:         ),
172:     )
173:
174:     val scaled = factory.makeMatrix(
175:         listOf(
176:             listOf(.10, .20, .40, .05, .10),
177:             listOf(.00, .10, .00, .20, .40),
178:             listOf(.10, .00, .10, .20, .40),
179:             listOf(.20, .00, .10, .10, .10),
180:         ),
181:     )
182:
183:     assertEquals(scaled, 0.1 * m1)
184: }
185:
186: @Test
187: fun 'right multiply by scalar'() {
188:     val m1 = factory.makeMatrix(
189:         listOf(
190:             listOf(1.0, 2.0, 4.0, 0.5, 1.0),
191:             listOf(0.0, 1.0, 0.0, 2.0, 4.0),
192:             listOf(1.0, 0.0, 1.0, 2.0, 4.0),
193:             listOf(2.0, 0.0, 1.0, 1.0, 1.0),
194:         ),
195:     )

```

```

196:
197:     val scaled = factory.makeMatrix(
198:         listOf(
199:             listOf(.10, .20, .40, .05, .10),
200:             listOf(.00, .10, .00, .20, .40),
201:             listOf(.10, .00, .10, .20, .40),
202:             listOf(.20, .00, .10, .10, .10),
203:         ),
204:     )
205:
206:     assertEquals(scaled, m1 * 0.1)
207: }
208:
209: @Test
210: fun 'string representation'() {
211:     val m1 = factory.makeMatrix(
212:         listOf(
213:             listOf(1.46, 2.0, 4.0, 0.5, 1.0),
214:             listOf(0.0, 1.0, 100.0, 2.0, 4.0),
215:             listOf(1.0, 0.0, 1.0, 2020.12, 4.0),
216:             listOf(2.0, 0.0, 1.0, 1.0, 1.0),
217:         ),
218:     )
219:
220:     val stringRepresentation =
221:         """
222:             [ 1.46 2.0   4.0     0.5 1.0 ]
223:             [  0.0 1.0 100.0    2.0 4.0 ]
224:             [  1.0 0.0   1.0 2020.12 4.0 ]
225:             [  2.0 0.0   1.0    1.0 1.0 ]
226:         """
227:     assertEquals(stringRepresentation, m1.toString())
228: }
229:
230: @Test
231: fun 'exception - empty matrix'() {
232:     try {
233:         factory.makeMatrix(emptyList())
234:     } catch (exception: IllegalArgumentException) {
235:         // Good: exception was expected.
236:     }
237: }
238:
239: @Test
240: fun 'exception - negative row index'() {
241:     try {
242:         factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0,
243: 1.0))).getRow(-1)
244:     } catch (exception: IndexOutOfBoundsException) {
245:         // Good: exception was expected.
246:     }
247: }
248:
249: @Test
250: fun 'exception - negative row index with operator'() {
251:     try {
252:         factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0, 1.0)))[-1]
253:     } catch (exception: IndexOutOfBoundsException) {
254:         // Good: exception was expected.
255:     }
256: }
257:
258: @Test
259: fun 'exception - negative column index'() {
260:     try {
261:         factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0,

```

```

1.0)).getColumn(-1)
263:         fail("IndexOutOfBoundsException was expected.")
264:     } catch (exception: IndexOutOfBoundsException) {
265:         // Good: exception was expected.
266:     }
267: }
268:
269: @Test
270: fun 'exception - too large row index'() {
271:     try {
272:         factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0, 1.0))).getRow(2)
273:         fail("IndexOutOfBoundsException was expected.")
274:     } catch (exception: IndexOutOfBoundsException) {
275:         // Good: exception was expected.
276:     }
277: }
278:
279: @Test
280: fun 'exception - too large row index with operator'() {
281:     try {
282:         factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0, 1.0)))[2]
283:         fail("IndexOutOfBoundsException was expected.")
284:     } catch (exception: IndexOutOfBoundsException) {
285:         // Good: exception was expected.
286:     }
287: }
288:
289: @Test
290: fun 'exception - too large column index'() {
291:     try {
292:         factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0,
1.0))).getColumn(2)
293:         fail("IndexOutOfBoundsException was expected.")
294:     } catch (exception: IndexOutOfBoundsException) {
295:         // Good: exception was expected.
296:     }
297: }
298:
299: @Test
300: fun 'exception - indices out of bounds'() {
301:     val m = factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0, 1.0)))
302:     try {
303:         val entry = m[-1, 1]
304:         fail("IndexOutOfBoundsException was expected.")
305:     } catch (exception: IndexOutOfBoundsException) {
306:         // Good: exception was expected.
307:     }
308:     try {
309:         val entry = m[1, -1]
310:         fail("IndexOutOfBoundsException was expected.")
311:     } catch (exception: IndexOutOfBoundsException) {
312:         // Good: exception was expected.
313:     }
314:     try {
315:         val entry = m[2, 0]
316:         fail("IndexOutOfBoundsException was expected.")
317:     } catch (exception: IndexOutOfBoundsException) {
318:         // Good: exception was expected.
319:     }
320:     try {
321:         val entry = m[0, 2]
322:         fail("IndexOutOfBoundsException was expected.")
323:     } catch (exception: IndexOutOfBoundsException) {
324:         // Good: exception was expected.
325:     }
326: }
327:
328: @Test

```

```

329:     fun 'exception - add matrices with different row counts'() {
330:         val m1 = factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0, 1.0)))
331:         val m2 = factory.makeMatrix(listOf(listOf(1.0, 1.0)))
332:         try {
333:             m1 + m2
334:             fail("UnsupportedOperationException was expected")
335:         } catch (exception: UnsupportedOperationException) {
336:             // Good: exception was expected.
337:         }
338:     }
339:
340: @Test
341: fun 'exception - add matrices with different column counts'() {
342:     val m1 = factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0, 1.0)))
343:     val m2 = factory.makeMatrix(listOf(listOf(1.0, 1.0, 1.0), listOf(1.0, 1.0,
1.0)))
344:     try {
345:         m1 + m2
346:         fail("UnsupportedOperationException was expected")
347:     } catch (exception: UnsupportedOperationException) {
348:         // Good: exception was expected.
349:     }
350: }
351:
352: @Test
353: fun 'exception - multiply matrices with incompatible sizes'() {
354:     val m1 = factory.makeMatrix(listOf(listOf(1.0, 1.0), listOf(1.0, 1.0)))
355:     val m2 = factory.makeMatrix(
356:         listOf(
357:             listOf(1.0, 1.0, 1.0),
358:             listOf(1.0, 1.0, 1.0),
359:             listOf(1.0, 1.0, 1.0),
360:         ),
361:     )
362:     try {
363:         m1 * m2
364:         fail("UnsupportedOperationException was expected")
365:     } catch (exception: UnsupportedOperationException) {
366:         // Good: exception was expected.
367:     }
368: }
369:
370: @Test
371: fun 'exception - create matrix with mismatched column counts'() {
372:     try {
373:         factory.makeMatrix(
374:             listOf(
375:                 listOf(1.0, 1.0),
376:                 listOf(1.0, 1.0, 1.0),
377:             ),
378:         )
379:         fail("IllegalArgumentException was expected")
380:     } catch (exception: IllegalArgumentException) {
381:         // Good: exception was expected.
382:     }
383: }
384: }

```

```

1: package algebra.generic
2:
3: import kotlin.test.Test
4: import kotlin.test.assertEquals
5: import kotlin.test.fail
6:
7: class DoubleVectorTests {
8:
9:     private val factory = AlgebraFactory(Double::plus, Double::times)
10:
11:     @Test
12:     fun 'vector length'() {
13:         val v = factory.makeVector(listOf(1.0, 2.0, 3.0))
14:         assertEquals(3, v.length)
15:     }
16:
17:     @Test
18:     fun 'get from vector'() {
19:         val v = factory.makeVector(listOf(1.0, 2.0, 3.0))
20:         assertEquals(1.0, v[0])
21:         assertEquals(2.0, v[1])
22:         assertEquals(3.0, v[2])
23:     }
24:
25:     @Test
26:     fun 'vector addition'() {
27:         val v1 = factory.makeVector(listOf(1.0, 2.0, 3.0))
28:         val v2 = factory.makeVector(listOf(4.0, 5.0, 6.0))
29:         val sum = factory.makeVector(listOf(5.0, 7.0, 9.0))
30:         assertEquals(sum, v1 + v2)
31:     }
32:
33:     @Test
34:     fun 'scalar times vector'() {
35:         val v1 = factory.makeVector(listOf(1.0, 2.0, 3.0))
36:         val scaled = factory.makeVector(listOf(10.0, 20.0, 30.0))
37:         assertEquals(scaled, v1 * 10.0)
38:     }
39:
40:     @Test
41:     fun 'vector times scalar'() {
42:         val v1 = factory.makeVector(listOf(1.0, 2.0, 3.0))
43:         val scaled = factory.makeVector(listOf(10.0, 20.0, 30.0))
44:         assertEquals(scaled, 10.0 * v1)
45:     }
46:
47:     @Test
48:     fun 'dot product'() {
49:         val v1 = factory.makeVector(listOf(1.0, 0.0, 0.0))
50:         val v2 = factory.makeVector(listOf(0.0, 1.0, 0.0))
51:         assertEquals(0.0, v1 dot v2)
52:     }
53:
54:     @Test
55:     fun 'dot product larger vectors'() {
56:         val v1 = factory.makeVector(listOf(1.0, 2.0, 3.0, 4.0, 5.0))
57:         val v2 = factory.makeVector(listOf(6.0, 7.0, 8.0, 9.0, 10.0))
58:         assertEquals(130.0, v1 dot v2)
59:     }
60:
61:     @Test
62:     fun 'string representation'() {
63:         val v1 = factory.makeVector(listOf(1.0, 2.0, 3.0, 4.0, 5.0))
64:         assertEquals("(1.0, 2.0, 3.0, 4.0, 5.0)", v1.toString())
65:     }
66:
67:     @Test
68:     fun 'exception - empty vector'() {

```

```

69:         try {
70:             factory.makeVector(emptyList())
71:             fail("IllegalArgumentException was expected.")
72:         } catch (exception: IllegalArgumentException) {
73:             // Good: exception was expected.
74:         }
75:     }
76:
77:     @Test
78:     fun 'exception - lengths do not match in addition'() {
79:         try {
80:             factory.makeVector(listOf(1.0, 2.0)) + factory.makeVector(listOf(1.0,
2.0, 3.0))
81:             fail("UnsupportedOperationException was expected.")
82:         } catch (exception: UnsupportedOperationException) {
83:             // Good: exception was expected.
84:         }
85:     }
86:
87:     @Test
88:     fun 'exception - lengths do not match in dot product'() {
89:         try {
90:             factory.makeVector(listOf(1.0, 2.0)) dot factory.makeVector(listOf(1.0,
2.0, 3.0))
91:             fail("UnsupportedOperationException was expected.")
92:         } catch (exception: UnsupportedOperationException) {
93:             // Good: exception was expected.
94:         }
95:     }
96:
97:     @Test
98:     fun 'exception - get at negative index'() {
99:         try {
100:             factory.makeVector(listOf(1.0, 2.0))[-1]
101:             fail("IndexOutOfBoundsException was expected.")
102:         } catch (exception: IndexOutOfBoundsException) {
103:             // Good: exception was expected.
104:         }
105:     }
106:
107:     @Test
108:     fun 'exception - get at too large index'() {
109:         try {
110:             factory.makeVector(listOf(1.0, 2.0))[2]
111:             fail("IndexOutOfBoundsException was expected.")
112:         } catch (exception: IndexOutOfBoundsException) {
113:             // Good: exception was expected.
114:         }
115:     }
116: }

```



```

1: package algebra.generic
2:
3: import kotlin.test.Test
4: import kotlin.test.assertEquals
5:
6: class MatrixExtensionTests {
7:
8:     @Test
9:     fun 'iterate over varargs-constructed matrix'() {
10:         val plus: (Int, Int) -> Int = { a, b -> a + b }
11:         val times: (Int, Int) -> Int = { a, b -> a * b }
12:
13:         val m1 = Matrix<Int>(
14:             plus,
15:             times,
16:             Vector(plus, times, listOf(10, 20, 30, 5, 10)),
17:             Vector(plus, times, listOf(0, 10, 0, 20, 30)),
18:             Vector(plus, times, listOf(10, 0, 10, 20, 40)),
19:             Vector(plus, times, listOf(20, 0, 10, 10, 10)),
20:         )
21:         val rows = mutableListof<Vector<Int>>()
22:         for (row in m1) {
23:             rows.add(row)
24:         }
25:         val expectedRows = listOf(
26:             Vector(plus, times, listOf(10, 20, 30, 5, 10)),
27:             Vector(plus, times, listOf(0, 10, 0, 20, 30)),
28:             Vector(plus, times, listOf(10, 0, 10, 20, 40)),
29:             Vector(plus, times, listOf(20, 0, 10, 10, 10)),
30:         )
31:         assertEquals(expectedRows, rows)
32:     }
33:
34:     @Test
35:     fun 'get element'() {
36:         val plus: (Int, Int) -> Int = { a, b -> a + b }
37:         val times: (Int, Int) -> Int = { a, b -> a * b }
38:
39:         val m1 = Matrix(
40:             plus,
41:             times,
42:             Vector(plus, times, listOf(10, 20, 30, 5, 10)),
43:             Vector(plus, times, listOf(0, 10, 0, 20, 30)),
44:             Vector(plus, times, listOf(10, 0, 10, 20, 40)),
45:             Vector(plus, times, listOf(20, 0, 10, 10, 10)),
46:         )
47:         assertEquals(10, m1[0, 0])
48:         assertEquals(20, m1[0, 1])
49:         assertEquals(30, m1[0, 2])
50:         assertEquals(5, m1[0, 3])
51:         assertEquals(10, m1[0, 4])
52:         assertEquals(0, m1[1, 0])
53:         assertEquals(10, m1[1, 1])
54:         assertEquals(0, m1[1, 2])
55:         assertEquals(20, m1[1, 3])
56:         assertEquals(30, m1[1, 4])
57:         assertEquals(10, m1[2, 0])
58:         assertEquals(0, m1[2, 1])
59:         assertEquals(10, m1[2, 2])
60:         assertEquals(20, m1[2, 3])
61:         assertEquals(40, m1[2, 4])
62:         assertEquals(20, m1[3, 0])
63:         assertEquals(0, m1[3, 1])
64:         assertEquals(10, m1[3, 2])
65:         assertEquals(10, m1[3, 3])
66:         assertEquals(10, m1[3, 4])
67:     }
68: }

```

```

1: package algebra.generic
2:
3: import kotlin.test.Test
4: import kotlin.test.assertEquals
5:
6: class NestedMatrixTests {
7:
8:     private val innerFactory = AlgebraFactory<Int>::plus, Int::times)
9:
10:     private val outerFactory = AlgebraFactory<Matrix<Int>::plus, Matrix<Int>::times)
11:
12:     @Test
13:     fun 'add nested matrices'() {
14:         val allZeroes2x2 = innerFactory.makeMatrix(
15:             listOf(
16:                 listOf(0, 0),
17:                 listOf(0, 0),
18:             ),
19:         )
20:
21:         val allOnes2x2 = innerFactory.makeMatrix(
22:             listOf(
23:                 listOf(1, 1),
24:                 listOf(1, 1),
25:             ),
26:         )
27:
28:         val m1 = outerFactory.makeMatrix(
29:             listOf(
30:                 listOf(allZeroes2x2, allOnes2x2),
31:                 listOf(allOnes2x2, allZeroes2x2),
32:             ),
33:         )
34:         val m2 = outerFactory.makeMatrix(
35:             listOf(
36:                 listOf(allOnes2x2, allZeroes2x2),
37:                 listOf(allZeroes2x2, allOnes2x2),
38:             ),
39:         )
40:         val expectedSum = outerFactory.makeMatrix(
41:             listOf(
42:                 listOf(allOnes2x2, allOnes2x2),
43:                 listOf(allOnes2x2, allOnes2x2),
44:             ),
45:         )
46:         assertEquals(expectedSum, m1 + m2)
47:     }
48:
49:     private fun make2x2Diagonal(value: Int): Matrix<Int> =
50:         innerFactory.makeMatrix(
51:             listOf(
52:                 listOf(value, 0),
53:                 listOf(0, value),
54:             ),
55:         )
56:
57:     @Test
58:     fun 'multiply nested matrices'() {
59:         val intMatrix1 = innerFactory.makeMatrix(
60:             listOf(
61:                 listOf(1, 2),
62:                 listOf(3, 4),
63:             ),
64:         )
65:
66:         val intMatrix2 = innerFactory.makeMatrix(
67:             listOf(
68:                 listOf(5, 6),

```

```

69:         listOf(7, 8),
70:     ),
71: )
72:
73: val intMatrixProduct1 = innerFactory.makeMatrix(
74:     listOf(
75:         listOf(19, 22),
76:         listOf(43, 50),
77:     ),
78: )
79:
80: val intMatrixProduct2 = innerFactory.makeMatrix(
81:     listOf(
82:         listOf(23, 34),
83:         listOf(31, 46),
84:     ),
85: )
86:
87: assertEquals(intMatrixProduct1, intMatrix1 * intMatrix2)
88:
89: assertEquals(intMatrixProduct2, intMatrix2 * intMatrix1)
90:
91: val nestedMatrix1 = outerFactory.makeMatrix(
92:     listOf(
93:         listOf(make2x2Diagonal(1), make2x2Diagonal(2)),
94:         listOf(make2x2Diagonal(3), make2x2Diagonal(4)),
95:     ),
96: )
97:
98: val nestedMatrix2 = outerFactory.makeMatrix(
99:     listOf(
100:         listOf(make2x2Diagonal(5), make2x2Diagonal(6)),
101:         listOf(make2x2Diagonal(7), make2x2Diagonal(8)),
102:     ),
103: )
104:
105: val nestedMatrixProduct1 = outerFactory.makeMatrix(
106:     listOf(
107:         listOf(make2x2Diagonal(19), make2x2Diagonal(22)),
108:         listOf(make2x2Diagonal(43), make2x2Diagonal(50)),
109:     ),
110: )
111:
112: val nestedMatrixProduct2 = outerFactory.makeMatrix(
113:     listOf(
114:         listOf(make2x2Diagonal(23), make2x2Diagonal(34)),
115:         listOf(make2x2Diagonal(31), make2x2Diagonal(46)),
116:     ),
117: )
118:
119: assertEquals(nestedMatrixProduct1, nestedMatrix1 * nestedMatrix2)
120:
121: assertEquals(nestedMatrixProduct2, nestedMatrix2 * nestedMatrix1)
122: }
123:
124: @Test
125: fun 'left and right multiplication by matrix scalar'() {
126:     val m1 = innerFactory.makeMatrix(
127:         listOf(
128:             listOf(1, 2),
129:             listOf(1, 2),
130:         ),
131:     )
132:
133:     val m2 = innerFactory.makeMatrix(
134:         listOf(
135:             listOf(1, 3),
136:             listOf(1, 3),

```

```

137:     ),
138: )
139:
140: val m1xm1 = innerFactory.makeMatrix(
141:     listOf(
142:         listOf(3, 6),
143:         listOf(3, 6),
144:     ),
145: )
146:
147: val m2xm2 = innerFactory.makeMatrix(
148:     listOf(
149:         listOf(4, 12),
150:         listOf(4, 12),
151:     ),
152: )
153:
154: val m1xm2 = innerFactory.makeMatrix(
155:     listOf(
156:         listOf(3, 9),
157:         listOf(3, 9),
158:     ),
159: )
160:
161: val m2xm1 = innerFactory.makeMatrix(
162:     listOf(
163:         listOf(4, 8),
164:         listOf(4, 8),
165:     ),
166: )
167:
168: assertEquals(m1xm1, m1 * m1)
169: assertEquals(m2xm2, m2 * m2)
170: assertEquals(m1xm2, m1 * m2)
171: assertEquals(m2xm1, m2 * m1)
172:
173: val nestedMatrix = outerFactory.makeMatrix(
174:     listOf(
175:         listOf(m1, m2, m1, m2),
176:         listOf(m2, m1, m2, m1),
177:     ),
178: )
179:
180: val nestedMatrixLeftScaledByM1 = outerFactory.makeMatrix(
181:     listOf(
182:         listOf(m1xm1, m1xm2, m1xm1, m1xm2),
183:         listOf(m1xm2, m1xm1, m1xm2, m1xm1),
184:     ),
185: )
186:
187: val nestedMatrixRightScaledByM1 = outerFactory.makeMatrix(
188:     listOf(
189:         listOf(m1xm1, m2xm1, m1xm1, m2xm1),
190:         listOf(m2xm1, m1xm1, m2xm1, m1xm1),
191:     ),
192: )
193:
194: val nestedMatrixLeftScaledByM2 = outerFactory.makeMatrix(
195:     listOf(
196:         listOf(m2xm1, m2xm2, m2xm1, m2xm2),
197:         listOf(m2xm2, m2xm1, m2xm2, m2xm1),
198:     ),
199: )
200:
201: val nestedMatrixRightScaledByM2 = outerFactory.makeMatrix(
202:     listOf(
203:         listOf(m1xm2, m2xm2, m1xm2, m2xm2),
204:         listOf(m2xm2, m1xm2, m2xm2, m1xm2),

```

```

205:         ),
206:     ),
207:
208:     assertEquals(nestedMatrixLeftScaledByM1, m1 * nestedMatrix)
209:     assertEquals(nestedMatrixRightScaledByM1, nestedMatrix * m1)
210:     assertEquals(nestedMatrixLeftScaledByM2, m2 * nestedMatrix)
211:     assertEquals(nestedMatrixRightScaledByM2, nestedMatrix * m2)
212: }
213: }

```

```

1: package algebra.generic
2:
3: import org.junit.Test
4: import kotlin.test.assertEquals
5:
6: class StringMatrixes {
7:
8:     @Test
9:     fun 'mutlitleply string matrices'() {
10:         val stringAlgebraFactory = AlgebraFactory<String>{
11:             plus = { a, b -> a + "+" + b },
12:             times = { a, b -> a + "*" + b },
13:         }
14:
15:         val m1 = stringAlgebraFactory.makeMatrix(
16:             listOf(
17:                 listOf("ant", "bug", "croc"),
18:                 listOf("deer", "elephant", "frog"),
19:             ),
20:         )
21:
22:         val m2 = stringAlgebraFactory.makeMatrix(
23:             listOf(
24:                 listOf("wasp", "beetle"),
25:                 listOf("goblin", "midge"),
26:                 listOf("mite", "kangaroo"),
27:             ),
28:         )
29:
30:         val product = stringAlgebraFactory.makeMatrix(
31:             listOf(
32:                 listOf("ant*wasp+bug*goblin+croc*mite",
33:                     "ant*beetle+bug*midge+croc*kangaroo"),
34:                 listOf("deer*wasp+elephant*goblin+frog*mite",
35:                     "deer*beetle+elephant*midge+frog*kangaroo"),
36:             ),
37:         )
38:
39:         assertEquals(product, m1 * m2)
40:
41:         val expectedString =
42:             """
43:             [ ant*wasp+bug*goblin+croc*mite
44:               ant*beetle+bug*midge+croc*kangaroo ]
45:             [ deer*wasp+elephant*goblin+frog*mite
46:               deer*beetle+elephant*midge+frog*kangaroo ]
47:             """.trimIndent()
48:
49:         assertEquals(expectedString, product.toString())
50:     }
51: }

```

```

1: package algebra.generic
2:
3: import kotlin.test.Test
4: import kotlin.test.assertEquals
5:
6: class VectorExtensionTests {
7:
8:     @Test
9:     fun 'iterate over varargs-constructed vector'() {
10:         val v = Vector({ s, t -> "$s+$t" }, { s, t -> "$s*$t" }, "a", "b", "c",
"d", "e")
11:         val elements = mutableListOf<String>()
12:         for (element in v) {
13:             elements.add(element)
14:         }
15:         assertEquals(listOf("a", "b", "c", "d", "e"), elements)
16:     }
17:
18:     @Test
19:     fun 'vector length'() {
20:         val v = Vector({ s, t -> "$s+$t" }, { s, t -> "$s*$t" }, "a", "b", "c",
"d", "e")
21:         assertEquals(5, v.length)
22:     }
23:
24:     @Test
25:     fun 'get from vector'() {
26:         val v = Vector({ s, t -> "$s+$t" }, { s, t -> "$s*$t" }, "a", "b", "c",
"d", "e")
27:         assertEquals("a", v[0])
28:         assertEquals("b", v[1])
29:         assertEquals("c", v[2])
30:         assertEquals("d", v[3])
31:         assertEquals("e", v[4])
32:     }
33: }

```

```

1: package algebra.real
2:
3: import kotlin.test.Test
4: import kotlin.test.assertEquals
5:
6: class MatrixExtensionTests {
7:
8:     @Test
9:     fun 'iterate over varargs-constructed matrix'() {
10:         val m1 = Matrix(
11:             Vector(1.0, 2.0, 3.0, 0.5, 1.0),
12:             Vector(0.0, 1.0, 0.0, 2.0, 3.0),
13:             Vector(1.0, 0.0, 1.0, 2.0, 4.0),
14:             Vector(2.0, 0.0, 1.0, 1.0, 1.0),
15:         )
16:         val rows = mutableListOf<Vector>()
17:         for (row in m1) {
18:             rows.add(row)
19:         }
20:         val expectedRows = listOf(
21:             Vector(1.0, 2.0, 3.0, 0.5, 1.0),
22:             Vector(0.0, 1.0, 0.0, 2.0, 3.0),
23:             Vector(1.0, 0.0, 1.0, 2.0, 4.0),
24:             Vector(2.0, 0.0, 1.0, 1.0, 1.0),
25:         )
26:         assertEquals(expectedRows, rows)
27:     }
28:
29:     @Test
30:     fun 'get element'() {
31:         val m1 = Matrix(
32:             Vector(1.0, 2.0, 3.0, 0.5, 1.0),
33:             Vector(0.0, 1.0, 0.0, 2.0, 3.0),
34:             Vector(1.0, 0.0, 1.0, 2.0, 4.0),
35:             Vector(2.0, 0.0, 1.0, 1.0, 1.0),
36:         )
37:         assertEquals(1.0, m1[0, 0])
38:         assertEquals(2.0, m1[0, 1])
39:         assertEquals(3.0, m1[0, 2])
40:         assertEquals(0.5, m1[0, 3])
41:         assertEquals(1.0, m1[0, 4])
42:         assertEquals(0.0, m1[1, 0])
43:         assertEquals(1.0, m1[1, 1])
44:         assertEquals(0.0, m1[1, 2])
45:         assertEquals(2.0, m1[1, 3])
46:         assertEquals(3.0, m1[1, 4])
47:         assertEquals(1.0, m1[2, 0])
48:         assertEquals(0.0, m1[2, 1])
49:         assertEquals(1.0, m1[2, 2])
50:         assertEquals(2.0, m1[2, 3])
51:         assertEquals(4.0, m1[2, 4])
52:         assertEquals(2.0, m1[3, 0])
53:         assertEquals(0.0, m1[3, 1])
54:         assertEquals(1.0, m1[3, 2])
55:         assertEquals(1.0, m1[3, 3])
56:         assertEquals(1.0, m1[3, 4])
57:     }
58: }

```

```

1: package algebra.real
2:
3: import kotlin.test.Test
4: import kotlin.test.assertEquals
5: import kotlin.test.fail
6:
7: class MatrixTests {
8:
9:     @Test
10:     fun 'get row'() {
11:         val m1 = Matrix(
12:             listOf(
13:                 Vector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)),
14:                 Vector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)),
15:                 Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)),
16:                 Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
17:             ),
18:         )
19:         assertEquals(Vector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)), m1.getRow(0))
20:         assertEquals(Vector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)), m1.getRow(1))
21:         assertEquals(Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)), m1.getRow(2))
22:         assertEquals(Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)), m1.getRow(3))
23:
24:         assertEquals(Vector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)), m1[0])
25:         assertEquals(Vector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)), m1[1])
26:         assertEquals(Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)), m1[2])
27:         assertEquals(Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)), m1[3])
28:     }
29:
30:     @Test
31:     fun 'get column'() {
32:         val m1 = Matrix(
33:             listOf(
34:                 Vector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)),
35:                 Vector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)),
36:                 Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)),
37:                 Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
38:             ),
39:         )
40:         assertEquals(Vector(listOf(1.0, 0.0, 1.0, 2.0)), m1.getColumn(0))
41:         assertEquals(Vector(listOf(2.0, 1.0, 0.0, 0.0)), m1.getColumn(1))
42:         assertEquals(Vector(listOf(3.0, 0.0, 1.0, 1.0)), m1.getColumn(2))
43:         assertEquals(Vector(listOf(0.5, 2.0, 2.0, 1.0)), m1.getColumn(3))
44:         assertEquals(Vector(listOf(1.0, 3.0, 4.0, 1.0)), m1.getColumn(4))
45:     }
46:
47:     @Test
48:     fun 'get element'() {
49:         val m1 = Matrix(
50:             listOf(
51:                 Vector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)),
52:                 Vector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)),
53:                 Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)),
54:                 Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
55:             ),
56:         )
57:         assertEquals(1.0, m1[0, 0])
58:         assertEquals(2.0, m1[0, 1])
59:         assertEquals(3.0, m1[0, 2])
60:         assertEquals(0.50, m1[0, 3])
61:         assertEquals(1.0, m1[0, 4])
62:         assertEquals(0.0, m1[1, 0])
63:         assertEquals(1.0, m1[1, 1])
64:         assertEquals(0.0, m1[1, 2])
65:         assertEquals(2.0, m1[1, 3])
66:         assertEquals(3.0, m1[1, 4])
67:         assertEquals(1.0, m1[2, 0])
68:         assertEquals(0.0, m1[2, 1])

```

```

69:         assertEquals(1.0, m1[2, 2])
70:         assertEquals(2.0, m1[2, 3])
71:         assertEquals(4.0, m1[2, 4])
72:         assertEquals(2.0, m1[3, 0])
73:         assertEquals(0.0, m1[3, 1])
74:         assertEquals(1.0, m1[3, 2])
75:         assertEquals(1.0, m1[3, 3])
76:         assertEquals(1.0, m1[3, 4])
77:     }
78:
79:     @Test
80:     fun 'matrix multiplication simple'() {
81:         val m1 = Matrix(
82:             listOf(
83:                 Vector(listOf(1.0, 1.0)),
84:             ),
85:         )
86:         val m2 = Matrix(
87:             listOf(
88:                 Vector(listOf(1.0)),
89:                 Vector(listOf(1.0)),
90:             ),
91:         )
92:         val product = m1 * m2
93:         assertEquals(Matrix(listOf(Vector(listOf(2.0)))), product)
94:     }
95:
96:     @Test
97:     fun 'matrix multiplication'() {
98:         val m1 = Matrix(
99:             listOf(
100:                 Vector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)),
101:                 Vector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)),
102:                 Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)),
103:                 Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
104:             ),
105:         )
106:
107:         val m2 = Matrix(
108:             listOf(
109:                 Vector(listOf(2.0, 3.0)),
110:                 Vector(listOf(1.0, 2.0)),
111:                 Vector(listOf(4.0, 1.0)),
112:                 Vector(listOf(0.0, 1.0)),
113:                 Vector(listOf(1.0, 3.0)),
114:             ),
115:         )
116:
117:         val product = Matrix(
118:             listOf(
119:                 Vector(listOf(17.0, 13.5)),
120:                 Vector(listOf(4.0, 13.0)),
121:                 Vector(listOf(10.0, 18.0)),
122:                 Vector(listOf(9.0, 11.0)),
123:             ),
124:         )
125:
126:         assertEquals(product, m1 * m2)
127:     }
128:
129:     @Test
130:     fun 'matrix addition'() {
131:         val m1 = Matrix(
132:             listOf(
133:                 Vector(listOf(1.0, 2.0, 3.0, 0.5, 1.0)),
134:                 Vector(listOf(0.0, 1.0, 0.0, 2.0, 3.0)),
135:                 Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)),
136:                 Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),

```

```

137:         ),
138:     ),
139:
140:     val m2 = Matrix(
141:         listOf(
142:             Vector(listOf(11.0, 12.0, 13.0, 10.5, 11.0)),
143:             Vector(listOf(10.0, 11.0, 10.0, 12.0, 13.0)),
144:             Vector(listOf(11.0, 10.0, 11.0, 12.0, 14.0)),
145:             Vector(listOf(12.0, 10.0, 11.0, 11.0, 11.0)),
146:         ),
147:     )
148:
149:     val sum = Matrix(
150:         listOf(
151:             Vector(listOf(12.0, 14.0, 16.0, 11.0, 12.0)),
152:             Vector(listOf(10.0, 12.0, 10.0, 14.0, 16.0)),
153:             Vector(listOf(12.0, 10.0, 12.0, 14.0, 18.0)),
154:             Vector(listOf(14.0, 10.0, 12.0, 12.0, 12.0)),
155:         ),
156:     )
157:
158:     assertEquals(sum, m1 + m2)
159: }
160:
161: @Test
162: fun 'left multiply by scalar'() {
163:     val m1 = Matrix(
164:         listOf(
165:             Vector(listOf(1.0, 2.0, 4.0, 0.5, 1.0)),
166:             Vector(listOf(0.0, 1.0, 0.0, 2.0, 4.0)),
167:             Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)),
168:             Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
169:         ),
170:     )
171:
172:     val scaled = Matrix(
173:         listOf(
174:             Vector(listOf(.10, .20, .40, .05, .10)),
175:             Vector(listOf(.00, .10, .00, .20, .40)),
176:             Vector(listOf(.10, .00, .10, .20, .40)),
177:             Vector(listOf(.20, .00, .10, .10, .10)),
178:         ),
179:     )
180:
181:     assertEquals(scaled, 0.1 * m1)
182: }
183:
184: @Test
185: fun 'right multiply by scalar'() {
186:     val m1 = Matrix(
187:         listOf(
188:             Vector(listOf(1.0, 2.0, 4.0, 0.5, 1.0)),
189:             Vector(listOf(0.0, 1.0, 0.0, 2.0, 4.0)),
190:             Vector(listOf(1.0, 0.0, 1.0, 2.0, 4.0)),
191:             Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
192:         ),
193:     )
194:
195:     val scaled = Matrix(
196:         listOf(
197:             Vector(listOf(.10, .20, .40, .05, .10)),
198:             Vector(listOf(.00, .10, .00, .20, .40)),
199:             Vector(listOf(.10, .00, .10, .20, .40)),
200:             Vector(listOf(.20, .00, .10, .10, .10)),
201:         ),
202:     )
203:
204:     assertEquals(scaled, m1 * 0.1)

```

```

205:     }
206:
207:     @Test
208:     fun 'string representation'() {
209:         val m1 = Matrix(
210:             listOf(
211:                 Vector(listOf(1.46, 2.0, 4.0, 0.5, 1.0)),
212:                 Vector(listOf(0.0, 1.0, 100.0, 2.0, 4.0)),
213:                 Vector(listOf(1.0, 0.0, 1.0, 2020.12, 4.0)),
214:                 Vector(listOf(2.0, 0.0, 1.0, 1.0, 1.0)),
215:             ),
216:         )
217:         val stringRepresentation =
218:             """
219:             [ 1.46 2.0 4.0 0.5 1.0 ]
220:             [ 0.0 1.0 100.0 2.0 4.0 ]
221:             [ 1.0 0.0 1.0 2020.12 4.0 ]
222:             [ 2.0 0.0 1.0 1.0 1.0 ]
223:             """.trimIndent()
224:
225:         assertEquals(stringRepresentation, m1.toString())
226:     }
227:
228:     @Test
229:     fun 'exception - empty matrix'() {
230:         try {
231:             Matrix(emptyList())
232:             fail("IllegalArgumentException was expected.")
233:         } catch (exception: IllegalArgumentException) {
234:             // Good: exception was expected.
235:         }
236:     }
237:
238:     @Test
239:     fun 'exception - negative row index'() {
240:         try {
241:             Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0,
242: 1.0))))).getRow(-1)
243:             fail("IndexOutOfBoundsException was expected.")
244:         } catch (exception: IndexOutOfBoundsException) {
245:             // Good: exception was expected.
246:         }
247:     }
248:
249:     @Test
250:     fun 'exception - negative row index with operator'() {
251:         try {
252:             Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0))))[-1]
253:             fail("IndexOutOfBoundsException was expected.")
254:         } catch (exception: IndexOutOfBoundsException) {
255:             // Good: exception was expected.
256:         }
257:     }
258:
259:     @Test
260:     fun 'exception - negative column index'() {
261:         try {
262:             Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0,
263: 1.0))))).getColumn(-1)
264:             fail("IndexOutOfBoundsException was expected.")
265:         } catch (exception: IndexOutOfBoundsException) {
266:             // Good: exception was expected.
267:         }
268:     }
269:
270:     @Test
271:     fun 'exception - too large row index'() {
272:         try {

```

```

../solution/src/test/kotlin/algebra/real/MatrixTests.kt      Fri Jan 26 11:09:32 2024      5
271:         Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0,
1.0))))).getRow(2)
272:         fail("IndexOutOfBoundsException was expected.")
273:     } catch (exception: IndexOutOfBoundsException) {
274:         // Good: exception was expected.
275:     }
276: }
277:
278: @Test
279: fun 'exception - too large row index with operator'() {
280:     try {
281:         Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0))))[2]
282:         fail("IndexOutOfBoundsException was expected.")
283:     } catch (exception: IndexOutOfBoundsException) {
284:         // Good: exception was expected.
285:     }
286: }
287:
288: @Test
289: fun 'exception - too large column index'() {
290:     try {
291:         Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0,
1.0))))).getColumn(2)
292:         fail("IndexOutOfBoundsException was expected.")
293:     } catch (exception: IndexOutOfBoundsException) {
294:         // Good: exception was expected.
295:     }
296: }
297:
298: @Test
299: fun 'exception - indices out of bounds'() {
300:     val m = Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0))))
301:     try {
302:         val entry = m[-1, 1]
303:         fail("IndexOutOfBoundsException was expected.")
304:     } catch (exception: IndexOutOfBoundsException) {
305:         // Good: exception was expected.
306:     }
307:     try {
308:         val entry = m[1, -1]
309:         fail("IndexOutOfBoundsException was expected.")
310:     } catch (exception: IndexOutOfBoundsException) {
311:         // Good: exception was expected.
312:     }
313:     try {
314:         val entry = m[2, 0]
315:         fail("IndexOutOfBoundsException was expected.")
316:     } catch (exception: IndexOutOfBoundsException) {
317:         // Good: exception was expected.
318:     }
319:     try {
320:         val entry = m[0, 2]
321:         fail("IndexOutOfBoundsException was expected.")
322:     } catch (exception: IndexOutOfBoundsException) {
323:         // Good: exception was expected.
324:     }
325: }
326:
327: @Test
328: fun 'exception - add matrices with different row counts'() {
329:     val m1 = Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0))))
330:     val m2 = Matrix(listOf(Vector(listOf(1.0, 1.0))))
331:     try {
332:         m1 + m2
333:         fail("UnsupportedOperationException was expected")
334:     } catch (exception: UnsupportedOperationException) {
335:         // Good: exception was expected.
336:     }

```

```

../solution/src/test/kotlin/algebra/real/MatrixTests.kt      Fri Jan 26 11:09:32 2024      6
337:     }
338: }
339: @Test
340: fun 'exception - add matrices with different column counts'() {
341:     val m1 = Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0))))
342:     val m2 = Matrix(listOf(Vector(listOf(1.0, 1.0, 1.0)), Vector(listOf(1.0,
1.0, 1.0))))
343:     try {
344:         m1 + m2
345:         fail("UnsupportedOperationException was expected")
346:     } catch (exception: UnsupportedOperationException) {
347:         // Good: exception was expected.
348:     }
349: }
350:
351: @Test
352: fun 'exception - multiply matrices with incompatible sizes'() {
353:     val m1 = Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0))))
354:     val m2 = Matrix(
355:         listOf(
356:             Vector(listOf(1.0, 1.0, 1.0)),
357:             Vector(listOf(1.0, 1.0, 1.0)),
358:             Vector(listOf(1.0, 1.0, 1.0)),
359:         ),
360:     )
361:     try {
362:         m1 * m2
363:         fail("UnsupportedOperationException was expected")
364:     } catch (exception: UnsupportedOperationException) {
365:         // Good: exception was expected.
366:     }
367: }
368:
369: @Test
370: fun 'exception - create matrix with mismatched column counts'() {
371:     try {
372:         Matrix(listOf(Vector(listOf(1.0, 1.0)), Vector(listOf(1.0, 1.0, 1.0))))
373:         fail("IllegalArgumentException was expected")
374:     } catch (exception: IllegalArgumentException) {
375:         // Good: exception was expected.
376:     }
377: }
378: }

```

```

1: package algebra.real
2:
3: import kotlin.test.Test
4: import kotlin.test.assertEquals
5:
6: class VectorExtensionTests {
7:
8:     @Test
9:     fun 'iterate over varargs-constructed vector'() {
10:         val v1 = Vector(1.0, 2.0, 3.0, 4.0, 5.0)
11:         val elements = mutableListOf<Double>()
12:         for (element in v1) {
13:             elements.add(element)
14:         }
15:         assertEquals(listOf(1.0, 2.0, 3.0, 4.0, 5.0), elements)
16:     }
17:
18:     @Test
19:     fun 'vector length'() {
20:         val v = Vector(1.0, 2.0, 3.0)
21:         assertEquals(3, v.length)
22:     }
23:
24:     @Test
25:     fun 'get from vector'() {
26:         val v = Vector(1.0, 2.0, 3.0)
27:         assertEquals(1.0, v[0])
28:         assertEquals(2.0, v[1])
29:         assertEquals(3.0, v[2])
30:     }
31: }

```

```

1: package algebra.real
2:
3: import kotlin.test.Test
4: import kotlin.test.assertEquals
5: import kotlin.test.fail
6:
7: class VectorTests {
8:
9:     @Test
10:    fun 'vector length'() {
11:        val v = Vector(listOf(1.0, 2.0, 3.0))
12:        assertEquals(3, v.length)
13:    }
14:
15:    @Test
16:    fun 'get from vector'() {
17:        val v = Vector(listOf(1.0, 2.0, 3.0))
18:        assertEquals(1.0, v[0])
19:        assertEquals(2.0, v[1])
20:        assertEquals(3.0, v[2])
21:    }
22:
23:    @Test
24:    fun 'vector addition'() {
25:        val v1 = Vector(listOf(1.0, 2.0, 3.0))
26:        val v2 = Vector(listOf(4.0, 5.0, 6.0))
27:        val sum = Vector(listOf(5.0, 7.0, 9.0))
28:        assertEquals(sum, v1 + v2)
29:    }
30:
31:    @Test
32:    fun 'scalar times vector'() {
33:        val v1 = Vector(listOf(1.0, 2.0, 3.0))
34:        val scaled = Vector(listOf(10.0, 20.0, 30.0))
35:        assertEquals(scaled, v1 * 10.0)
36:    }
37:
38:    @Test
39:    fun 'vector times scalar'() {
40:        val v1 = Vector(listOf(1.0, 2.0, 3.0))
41:        val scaled = Vector(listOf(10.0, 20.0, 30.0))
42:        assertEquals(scaled, 10.0 * v1)
43:    }
44:
45:    @Test
46:    fun 'dot product'() {
47:        val v1 = Vector(listOf(1.0, 0.0, 0.0))
48:        val v2 = Vector(listOf(0.0, 1.0, 0.0))
49:        assertEquals(0.0, v1 dot v2)
50:    }
51:
52:    @Test
53:    fun 'dot product larger vectors'() {
54:        val v1 = Vector(listOf(1.0, 2.0, 3.0, 4.0, 5.0))
55:        val v2 = Vector(listOf(6.0, 7.0, 8.0, 9.0, 10.0))
56:        assertEquals(130.0, v1 dot v2)
57:    }
58:
59:    @Test
60:    fun 'string representation'() {
61:        val v1 = Vector(listOf(1.0, 2.0, 3.0, 4.0, 5.0))
62:        assertEquals("(1.0, 2.0, 3.0, 4.0, 5.0)", v1.toString())
63:    }
64:
65:    @Test
66:    fun 'exception - empty vector'() {
67:        try {
68:            Vector(emptyList())

```



```

69:         fail("IllegalArgumentException was expected.")
70:     } catch (exception: IllegalArgumentException) {
71:         // Good: exception was expected.
72:     }
73: }
74:
75: @Test
76: fun 'exception - lengths do not match in addition'() {
77:     try {
78:         Vector(listOf(1.0, 2.0)) + Vector(listOf(1.0, 2.0, 3.0))
79:         fail("UnsupportedOperationException was expected.")
80:     } catch (exception: UnsupportedOperationException) {
81:         // Good: exception was expected.
82:     }
83: }
84:
85: @Test
86: fun 'exception - lengths do not match in dot product'() {
87:     try {
88:         Vector(listOf(1.0, 2.0)) dot Vector(listOf(1.0, 2.0, 3.0))
89:         fail("UnsupportedOperationException was expected.")
90:     } catch (exception: UnsupportedOperationException) {
91:         // Good: exception was expected.
92:     }
93: }
94:
95: @Test
96: fun 'exception - get at negative index'() {
97:     try {
98:         Vector(listOf(1.0, 2.0))[-1]
99:         fail("IndexOutOfBoundsException was expected.")
100:    } catch (exception: IndexOutOfBoundsException) {
101:        // Good: exception was expected.
102:    }
103: }
104:
105: @Test
106: fun 'exception - get at too large index'() {
107:     try {
108:         Vector(listOf(1.0, 2.0))[2]
109:         fail("IndexOutOfBoundsException was expected.")
110:    } catch (exception: IndexOutOfBoundsException) {
111:        // Good: exception was expected.
112:    }
113: }
114: }

```