

FizzBuzz

```
private List<String> fizzBuzz(int count) {  
  
    var fizz = Stream.of("", "", "Fizz").cycle();  
    // var fizz = Stream.fill(2, constant("")).append("Fizz").cycle();  
  
    var buzz = Stream.of("", "", "", "", "Buzz").cycle();  
    // var buzz = Stream.fill(4, constant("")).append("Buzz").cycle();  
  
    var fizzBuzz = fizz.zipWith(buzz, (fst, snd) -> fst + snd);  
    return fizzBuzz  
        .zipWithIndex((fb, i) -> fb.isBlank() ? "" + (i + 1) : fb)  
        .take(count)  
        .toList();  
  
}
```

Game of Life

```
private List<Tuple2<Integer, Integer>> neighboursOf(Tuple2<Integer, Integer> cell) {  
    return List.of(-1, 0, 1).flatMap(dx -> List.of(-1, 0, 1).map(dy -> cell(dx, dy)))  
        .filter(it -> !it.equals(cell(0, 0)))  
        .map(it -> cell(cell._1 + it._1, cell._2 + it._2));  
}  
  
private List<Tuple2<Integer, Integer>> candidatesFor(List<Tuple2<Integer, Integer>> cells) {  
    return cells.flatMap(this::neighboursOf).prependAll(cells).distinct();  
}  
  
private int numberOfLivingNeighbours(Tuple2<Integer, Integer> cell, List<Tuple2<Integer, Integer>> livingCells) {  
    return neighboursOf(cell).filter(livingCells::contains).size();  
}  
  
private List<Tuple2<Integer, Integer>> evolve(List<Tuple2<Integer, Integer>> world) {  
    return candidatesFor(world).filter(it -> willLive(it, world));  
}
```

List 1

```
@Test
public void foldLeftTest() {
    Function1<List<Integer>, Integer> sum = xs -> xs.foldLeft(0, (s, x) -> s + x);
    assertThat(sum.apply(List.rangeClosed(1, 10))).isEqualTo(10/2 * (10 + 1));

    Function1<List<Integer>, Integer> prod = xs -> xs.foldLeft(1, (p, x) -> p * x);
    assertThat(prod.apply(List.rangeClosed(1, 10))).isEqualTo(3_628_800);

    Function1<List<Integer>, Integer> count = xs -> xs.foldLeft(0, (n, x) -> n + 1);
    assertThat(count.apply(List.rangeClosed(1, 10))).isEqualTo(10);

    Function1<List<Integer>, List<Integer>> reverse = xs -> xs.foldLeft(List.empty(), List::prepend);
    assertThat(reverse.apply(List.rangeClosed(1, 5))).isEqualTo(List.of(5, 4, 3, 2, 1));

    Function1<List<Boolean>, Boolean> and = bs -> bs.foldLeft(true, (p, b) -> p && b);
    assertThat(and.apply(List.of(true, true, true))).isEqualTo(true);
    assertThat(and.apply(List.of(true, true, false))).isEqualTo(false);

    Function1<List<Boolean>, Boolean> or = bs -> bs.foldLeft(false, (p, b) -> p || b);
    assertThat(or.apply(List.of(true, true, false))).isEqualTo(true);
    assertThat(or.apply(List.of(false, false, false))).isEqualTo(false);
}
```

List 2

```
@Test
public void qsortTest() {
    var random = new Random();
    var randomNumbers = List.range(0, 10).map(it -> random.nextInt());
    var sortedNumbers = qsort(randomNumbers);
    var pairs = sortedNumbers.zipWith(sortedNumbers.tail(), Tuple::of);
    assertThat(pairs).allMatch(pair -> pair._1 <= pair._2);
}

private List<Integer> qsort(List<Integer> xs) {
    if (xs.isEmpty()) {
        return List.empty();
    }
    var head = xs.head();
    var tail = xs.tail();
    var smaller = tail.filter(it -> it <= head);
    var larger = tail.filter(it -> it > head);
    return qsort(larger).prepend(head).prependAll(qsort(smaller));
}
```

Option 1

```
@Test
public void saveDivideTest() {
    Function2<Integer, Integer, Integer> unsafeDivide = (x, y) -> x / y;

    assertThatExceptionOfType(ArithmeticException.class).isThrownBy(() ->
        unsafeDivide.apply(1, 0)
    );

    // Function2<Integer, Integer, Option<Integer>>
    var saveDivide = Function2.lift(unsafeDivide);

    assertThat(saveDivide.apply(1, 0)).isEqualTo(Option.none());

    // Function1<Integer, Option<Integer>>
    var reciprocalPercent = saveDivide.apply(100);

    // List<Option<Integer>>
    var percentages = List.of(-2, -1, 0, 1, 2).map(reciprocalPercent);

    assertThat(percentages).isEqualTo(
        List.of(Option.some(-50), Option.some(-100), Option.none(), Option.some(100), Option.some(50))
    );

    assertThat(percentages.flatMap(Function1.identity())).isEqualTo(
        List.of(-50, -100, 100, 50)
    );
}
```

Option 2

```
@Test
public void optionalIsNotMonadic() {
    assertThat(Optional.of(2).map(y -> divideOrNull(10, y))).isEqualTo(Optional.of(5));
    assertThat(Optional.of(0).map(y -> divideOrNull(10, y))).isEqualTo(Optional.empty());

    var optionalResult = Optional.of(0).map(y -> divideOrNull(10, y));
    assertThat(optionalResult.map(it -> it * 10)).isEqualTo(Optional.empty());
}
```

```
@Test
public void optionIsMonadic() {
    assertThat(Option.some(2).map(y -> divideOrNull(10, y))).isEqualTo(Option.some(5));
    assertThat(Option.some(0).map(y -> divideOrNull(10, y))).isEqualTo(Option.some(null));

    var optionResult = Option.some(0).map(y1 -> divideOrNull(10, y1));
    assertThatNullPointerException().isThrownBy(() ->
        optionResult.map(it -> it * 10)
    );

    var optionResult2 = Option.some(0)
        .map(y1 -> divideOrNull(10, y1))
        .flatMap(result -> Option.of(result).map(it -> it * 10));
    assertThat(optionResult2).isEqualTo(Option.none());

    var optionResult3 = Option.some(0)
        .flatMap(y -> Option.of(divideOrNull(10, y)));
    assertThat(optionResult3).isEqualTo(Option.none());
}
```

Try 1

```
@Test
public void tryToDivide() {
    Function2<Integer, Integer, Integer> unsafeDivide = (x, y) -> x / y;
    // Function2<Integer, Integer, Try<Integer>> tryDivide = (x, y) -> Try.of(() -> unsafeDivide.apply(x, y));
    var tryDivide = Function2.liftTry(unsafeDivide);

    assertThat(tryDivide.apply(10, 2)).isEqualTo(Try.success(5));

    var failingDivision = tryDivide.apply(10, 0);
    assertThat(failingDivision.isFailure()).isTrue();
    assertThat(failingDivision.getCause()).isInstanceOf(ArithmeticException.class);
    assertThat(failingDivision.getOrElse(MAX_VALUE)).isEqualTo(MAX_VALUE);
    assertThat(failingDivision.recover(ArithmeticException.class, MAX_VALUE).get()).isEqualTo(MAX_VALUE);
}
```

Try 2

```
@Test
public void tryToReadFiles() {
    Function1<String, Try<List<String>>> tryReadLinesFrom = filename -> Try.of(() -> readLinesFrom(filename));

    var result = tryReadLinesFrom.apply("/first.txt")
        .mapTry(lines -> lines.get(1))
        .flatMapTry(tryReadLinesFrom::apply)
        .mapTry(lines -> lines.get(1))
        .onFailure(Throwable::printStackTrace);

    assertThat(result.getOrElse("FAILED")).isEqualTo("SUCCESS");
}
```