Zadanie E - Funkcje wieloargumentowe

Uwagi:

- do zaimplementowania klasa o nazwie Functions w pakiecie domyślnym i pliku `Functions.java`
- interfejs `Function` i klasa `GenericFunctionsException` będą zapewnione przy testach
- niedozwolone jest definiowanie klas o nazwach zaczynających się słowem Test

Funkcja, to odwzorowanie z pewnego zbioru w inny zbiór. W tym zadaniu funkcje są wieloargumentowe, a każda funkcja implementuje następujący interfejs:

```
public interface Function<T, S> {
    int arity();
    S compute(List<? extends T> args) throws GenericFunctionsException;
}
```

Aby skorzystać z funkcji należy wywołać ją (metodą compute) na liście argumentów długości **dokładnie odpowiadającej** arności funkcji. Przykładowa implementacja funkcji, w tym przypadku jedno-argumentowej wygląda tak:

```
import java.util.List;

public class StrRvs implements Function<String, String> {
    @Override
    public int arity() { return 1;     }
    @Override
    public String compute(List<? extends String> args) throws GenericFunctionsException {
        if (args == null || args.size() != arity()) throw new GenericFunctionsException();
        return new StringBuilder(args.get(0)).reverse().toString();
    }
}
```

Twoim zadaniem jest zaimplementowanie klasy Functions definiującej trzy statyczne publiczne metody:

- `constant(...)` produkującą funkcje stałe, czyli zero-argumentowe. Metoda ta
  - ma dwa parametry typu
  - bierze element odpowiedniego typu
  - zwraca funkcję zero-argumentową (odpowiedniego typu) której wynikiem jest zawsze element podany podczas konstrukcji
- `proj(...)` która produkuje funkcje będące projekcjami. Metoda ta
  - ma dwa parametry typu
  - bierze dwie liczby typu int: n oraz k
  - zwraca funkcję n-argumentową, której wynikiem wykonania jest k-ty spośród argumentów.

- `compose(...)`, która składa funkcje wieloargumentowe. Metoda ta
  - ma trzy parametry typu
  - bierze jako pierwszy argument funkcję zewnętrzną
  - bierze jako drugi argument listę funkcji wewnętrznych o tej samej arności (oraz odpowiednich dziedzinach i przeciwdziedzinach)
  - i produkuje (jeśli to możliwe) klasę odpowiadającą funkcji będącej złożeniem argumentów na przykład: dla funkcji dwu-argumentowych `f(x,y)`, `g(x,y)`, `h(x,y)` wywołanie `Functions.compose(f,g,h)` zwróci funkcję `f(g(x,y),h(x,y))` (patrz przykładowe testy).

Jeśli wyprodukowanie odpowiedniej funkcji jest niemożliwe powinien być rzucony wyjątek

```
@SuppressWarnings("serial")
public class GenericFunctionsException extends Exception { }
```

Uwaga plik Functions.java ma kompilować się przy użyciu `javac -Xlint Functions.java` bez **żadnych** ostrzeżeń, i nie zawierać adnotacji typu `@SuppressWarnings`.

Przykładowe testy

```
import java.util.Arrays;
import java.util.Collections;

@Test
public void test1() throws GenericFunctionsException {
    Function<String, String> f = Functions.constant("Ala");
    assertEquals("Ala", f.compute(Collections.<String> emptyList()));
    assertEquals((Double) 123.0, Functions.constant(123.0).compute(
            Collections.<Double> emptyList()));
    try {
        System.out.println(Functions.constant("Ala").compute(
                Arrays.asList("Ala")));
        fail();
    } catch (GenericFunctionsException e) {
        System.out.println("Incorrect number of arguments");
    }
}

@Test
public void test2() throws GenericFunctionsException {
    Function<Integer, Integer> f = Functions.proj(3, 0);
    assertEquals(1, (long) f.compute(Arrays.asList(1, 2, 3)));
    assertEquals("ma", Functions.<String, String> proj(2, 1).compute(
            Arrays.asList("Ala", "ma")));
    try {
        System.out.println(Functions.<String, String> proj(2, 1).compute(
                Arrays.asList("Ala")));
        fail();
```

```
    } catch(GenericFunctionsException e) {
        System.out.println("Too few arguments");
    }
}
```

Przykładowe implementacje funkcji:

```
import java.util.List;

public class StrRvs implements Function<String, String> {
    @Override
    public int arity() { return 1;     }
    @Override
    public String compute(List<? extends String> args) throws GenericFunctionsException {
        if (args == null || args.size() != arity()) throw new GenericFunctionsException();
        return new StringBuilder(args.get(0)).reverse().toString();
    }
}
```

```
import java.util.List;

public class StrConcat implements Function<String, String> {
    @Override
    public int arity() { return 2; }
    @Override
    public String compute(List<? extends String> args) throws GenericFunctionsException {
        if (args == null || args.size() != arity()) throw new GenericFunctionsException();
        return args.get(0).toString() + args.get(1);
    }
}
```

Kolejne testy korzystające z powyższych klas:

```
import java.util.Arrays;
import java.util.List;

@Test
public void test3() throws GenericFunctionsException {
    Function<String, String> f = new StrRvs();
    List<String> ala = Arrays.asList("Ala");
    assertEquals("alA",f.compute(ala));
    f = Functions.compose(f, Arrays.asList(f));
    assertEquals("Ala",f.compute(ala));
    f = Functions.compose(new StrRvs(), Arrays.asList(f));
    assertEquals("alA",f.compute(ala));
}

@Test
public void test4() throws GenericFunctionsException {
    Function<String, String> f = Functions.compose(new StrRvs(),
            Arrays.asList(new StrRvs()));
```

```
    f = Functions.compose(new StrConcat(), Arrays.asList(f, new StrRvs()));
    assertEquals("AlaalA",f.compute(Arrays.asList("Ala")));
    f = Functions
            .compose(new StrConcat(), Arrays.asList(new StrConcat(), new StrConcat()));
    assertEquals("****",f.compute(Arrays.asList("*","*")));
}

@Test
public void test5() throws GenericFunctionsException {
    Function<String, String> f = Functions.compose(new StrRvs(),
            Arrays.asList(new StrRvs()));
    f = Functions.compose(new StrConcat(), Arrays.asList(f, new StrRvs()));
    assertEquals("AlaalA",f.compute(Arrays.asList("Ala")));
    assertEquals("ma",Functions.proj(3, 1).compute(
            Arrays.asList("Ala", "ma", "kota")));
    f = Functions.compose(
            new StrConcat(),
            Arrays.asList(Functions.<String, String> proj(2, 0),
                    Functions.<String, String> proj(2, 1)));
    assertEquals("AlaMa",f.compute(Arrays.asList("Ala", "Ma")));
    f = Functions.compose(f, Arrays.asList(
            Functions.compose(f, Arrays.asList(
                    Functions.<String, String> proj(4, 0),
                    Functions.<String, String> proj(4, 1))),
            Functions.compose(f, Arrays.asList(
                    Functions.<String, String> proj(4, 2),
                    Functions.<String, String> proj(4, 3)))));
    assertEquals("AlaMaKota?",f.compute(Arrays.asList("Ala", "Ma", "Kota", "?")));
    f = Functions.compose(
            f,
            Arrays.asList(Functions.<String, String> proj(1, 0),
                    Functions.<String, String> proj(1, 0),
                    Functions.<String, String> proj(1, 0),
                    Functions.<String, String> proj(1, 0)));
    assertEquals("++++",f.compute(Arrays.asList("+")));
}
```