Object-Oriented Programming

Worksheet 3

Tom Lin Jamie Willis

February 1, 2021

1 Generics

1. Implement the generic **static** method `reverse` that, when given a *parameterised* (i.e has a type parameter, say `<T>`) `java.util.List`, will return a reversed list. The method should not mutate the original list but return a new collection of your choice from the *Collections Framework* that implements the `java.util.List` interface with the same generic type.

   Consider placing this method inside an *utility class* to facilitate compilation and testing.

2. Clearly define the following terms, give examples of each to support your definition:
   - Subtype polymorphism (subtyping)
   - Parametric polymorphism
   - Ad-hoc polymorphism

† 3. Java's parametric polymorphism supports arbitrary [1] numbers of type parameters. The arity of a generic class denotes how many type parameters a class has, for example, `java.lang.String` is a 0-ary class where as `java.lang.Map<K,V>` is a 2-ary class.

   (a) Implement the 1-ary immutable container class `Id`, this class should contain a constructor that takes one generic parameter and store it as a public final field.

   (b) Using the same structure as `Id`, implement the 2-ary and 3-ary variant named `Tuple` and `Triple` respectively.

   (c) Implement the 0-ary variant `Unit`, this class stores no values. Make sure all instances of this class are the referentially equal. Then explain the difference between `Unit`, `java.lang.Object`, `null`s and the `void` return type.

†† 4. Implement the 1-ary generic class `Maybe<T>`, similar to `java.util.Optional` or Haskell's `Maybe`, this class is either empty or contains a value. `Maybe` should conceal public constructors and offer static methods `<T> Maybe<T> nothing()` and `<T> Maybe<T> just(T)` to instantiate.

   Import classes `java.util.function.Function` and `java.util.function.Supplier` and review their documentation. Then implement following methods for `Maybe`:

   - `boolean isEmpty()` - whether this `Maybe` contains a value.
   - `<U> Maybe<U> map(Function<T, U>)` - transforms the current maybe value with the given function if non empty, nothing otherwise.
   - `<U> Maybe<U> flatMap(Function<T, Maybe<U> >)` - like `map` but flattens the maybe returned by the given function.
   - `<U> U fold(Function<T, U>, Supplier<U>)` - transforms the value using the given function if non empty or returns the given default from the supplier.
   - `T get()` - return the value if non empty or throws a `NoSuchElementException`.

[1] Dependent on the compiler, usually limited by the class file constant pool size.

2 Libraries

1. Graph traversal is a task that can be modelled well in an OOP language like Java. Nodes are simply objects that contain references to edges and so on.

   (a) Similar to *Makefile*s in C and *Cabal/Stack* in Haskell, *Maven* is one of the most common build system in Java.
   For this question, you will be developing in a Maven skeleton project, download the project on the unit page. You may use an IDE of your choice to open the project; the officially supported IDE is IntelliJ IDEA. Alternatively, your text editor and terminal will suffice.

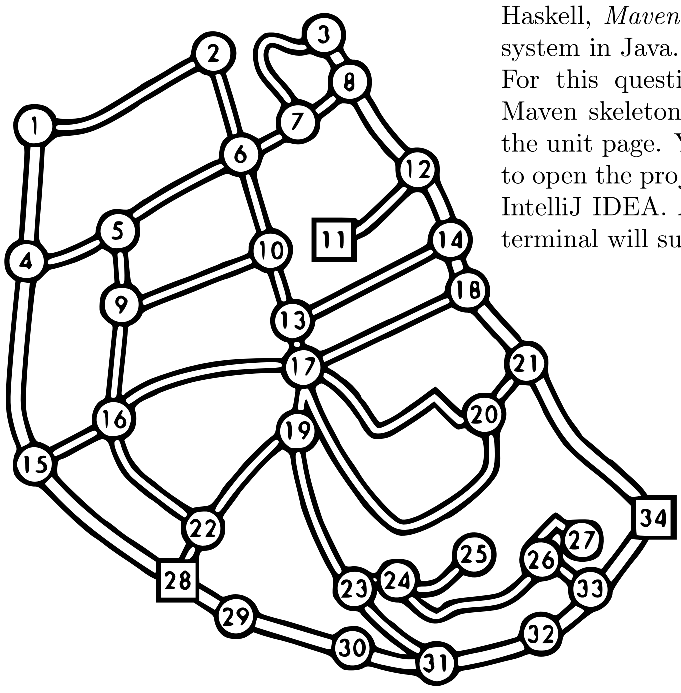

Figure 1: *University of Bristol* path map

   (b) Once the project archive is extracted, run the following command at project root:

   `./mvnw clean test` (Omit leading `./` for `mvnw` on Windows)

   The command will compile the project and run unit tests. As the tasks are unimplemented at this point, you will observe test failures. Identify the following sources files:

   - `<project_root>/src/main/java/Search.java` - the task skeleton
   - `<project_root>/src/main/resources/map.txt` - text representation of university map.
   - `<project_root>/src/test/java/SearchTest.java` - unit tests for the task

   Testing in done with the standard *JUnit4* library. Although you will only be reading the tests, knowing how the tests are structured is essential. Briefly go through the documentation to learn about unit testing. The graph structure uses Guava's graph module to represent undirected graphs with values on edges. Briefly skim the Graph guide. The graph used in this task is a `ValueGraph`, refer to the JavaDocs for API documentations.

   (c) Given a populated graph with nodes as numbered markers and edges the length of each road. Implement the `listAllNodes`, `listAllEdges` methods in `Search.java`. See the source JavaDocs on the exact requirements.
   Verify that these two methods pass the unit test by running: `./mvnw test -Dtest="#testList*"`

† (d) Implement the `findAllNodeWith4OrMoreEdges`, `findAllNodesWithEdgeSumGreaterThan20` methods in `Search.java`.
   Verify that these two methods pass the unit test by running: `./mvnw test -Dtest="#testFindAll*"`

† † † (e) Implement the `shortestPathFromSourceToDestination` method in `Search.java` using any shortest path first algorithm (e.g Dijkstras, A*) of your choice.
   Verify that it works by passing the unit test: `./mvnw test -Dtest="#testShortestPath"`