EECS 293
Software Craftsmanship
2020 Spring Semester

Programming Assignment 3

Due at the beginning of your discussion session on

February 4-7, 2020

Reading

In addition to the following topics, the quiz syllabus includes any material covered in the lectures:

- Section 16.1 ("Normal Loop-With-Exit Loops" only) in Code Complete
- Section 16.2 (Except "Enter the loop from one location only", "Checking Endpoints", "Using Loop Variables", and "How Long …") in Code Complete.
- Section 19.6 in Code Complete
- Item 58, 59, and 67 in Effective Java.

Programming

First, make all the changes discussed in your discussion section. Additionally, you should refactor your code to make sure that it adopts the principles covered in the reading assignments. In particular, you should modify your code to avoid methods with high McCabe's complexity.

In this programming assignment, you will implement an algorithm to match two type expressions.

Compound Types

Compound types, such as `List`, are meant to be used in conjunction with other types, such as in the case of `List<T>` or `List<Integer>`. Define a `public final class CompoundType` that implements `Type`. A `CompoundType` has a `private final TypeName typeName`.

It also has a `private final int arity` with a public getter. The arity represent the number of sub-types. For example, the arity of List is 1 (as in `List<Integer>`, where `Integer` is the sub-type), and the arity Map is 2 (as in `Map<K, V>`, where `K` and `V` are the sub-types). `CompoundType` has a private constructor that sets the `typeName` and the arity and cannot throw any exception. It has

```
public static final CompoundType of(
            String identifier, int arity)
```

that returns a new `CompoundType` with the corresponding type name identifier and arity. If the arity is not positive, it throws an `IllegalArgumentException` with an appropriate message.

The `CompoundType` delegates `getIdentifier` and `toString` to the `typeName`. It overrides `isVariable` to return false.

The `public final class ArityException` extends `Exception` to provide failure information in cases when the arity is not satisfied. It has two private final variable representing a compound type and a list of sub-types. These variables are initialized by the public constructor and accessible via public getters. Its `serialVersionUID` is required but unused in this project, and can be randomly generated by the IDE.

The `CompoundTypeEntry` is an entry corresponding to a compound type. A single compound type, such as `List`, can have multiple corresponding type entries, such as `List<Integer>`, `List<Boolean>`, `List<Double>`, and `List<T>`. `CompoundTypeEntry` extends `AbstractTypeEntry` and has `private final CompoundType type` with a public getter and a `private final List<TypeEntry> subTypes` with a getter that returns a copy of the sub-types list. `CompoundTypeEntry` has a private constructor that sets the `type` and `subTypes`, and cannot throw any exception. It also has a

```
public static final CompoundTypeEntry of(
      CompoundType type,
      List<TypeEntry> subTypes) throws ArityException
```

that returns a new `CompoundEntry` of the given type with a copy of the provided sub-types list. It throws an `ArityException` if the number of sub-types does not match the arity.

The `toString` method builds the string representation recursively from the sub-types. It consists of the `toString` of the type followed by a <, the `toString` of the sub-types separated by commas, and terminated by a >:

type-toString<sub-type1-toString, … , sub-typen-toString>

An example is: `Map<Integer, List<Double>>.`

Add to `TypeEntry` the method `List<TypeEntry> getSubTypes()`. A `CompoundTypeEntry` will provide a public implementation that returns a copy of the sub-types. This method should not be invoked on `SimpleTypeEntries` if the code is correct, but it can return an empty list.

Unification

Unification is the process of making two type expressions identical by substituting expressions for their variables. For example, if

```
<S,T> List<T> foo(Function<S,T> f, List<Integer> l) {
          return new ArrayList<>();
}
<U> bar(Function<Integer, U> f, List<Integer> l) {
          return foo(f, l);
}
```

then unification should substitute the return value of bar with `List<U>`. Unification is the critical step in type inference.

Unification relies on data structures that gives information on the equivalence of type systems. In the example, the data structure should state that `S` is equivalent to `Integer`. Create a package-private `final class TypeGroup`, which is meant to represent all the types that are equivalent to each other. The `TypeGroup` has a `private final Set<TypeEntry> typeGroup`, which are all the types in the group, and a `private TypeEntry representative`, which is a member of the group and functions as its canonical representative. Additional private variable will be added shortly. The `TypeGroup` has a private constructor that sets the `typeGroup` and its `representative` and

`static final TypeGroup of(TypeEntry typeEntry)`

which returns a new `TypeGroup`, containing only the given `typeEntry` (which is also its `representative`). The `TypeGroup` has a package-private getter for the representative. The package-private `size()` method is delegated to the `typeGroup`. `TypeGroup` implements `Iterable<TypeEntry>` and delegates the iterator to the `typeGroup`.

Create a `public final class TypeSystem`, which is meant to contain all the type groups. Then, modify `TypeGroup` so that it contains a reference to its `TypeSystem` by adding a `private final TypeSystem typeSystem`, adding one more argument to the constructor and to the `of` method. The `TypeSystem` has a private `Map<TypeEntry, TypeGroup> groups`, which is initially empty, and which maps a type entry with the group to which the type entry belongs.

The `TypeSystem` has `public final TypeEntry add(TypeEntry typeEntry)`, which creates a new `TypeGroup` containing only the `typeEntry` , correspondingly updates the `groups`, and returns the `typeEntry`. The method has no effect if the `typeEntry` already belongs to the type system.

The `TypeSystem` has a package-private `TypeEntry representative(TypeEntry s)` that returns the representative corresponding to the given type entry. If `s` is not in the type system, it throws an `IllegalStateException` whose cause is a `MissingTypeEntryException` with `s`. (The `MissingTypeEntryException` is similar to the `ArityException`, but it has `private final TypeEntry` and `private final TypeSystem` variables with getters.)

At this point, the `TypeSystem` contains the supporting elements for running the unification algorithm, which will be the topic of the next assignment.

General Considerations

These classes may contain as many auxiliary private and package-private methods as you see fit, and additional package-private helper classes may be defined. However, any modification or addition to public classes and methods must be approved by the instructors at the discussion board.

You should write JUnit tests to make sure that your primary methods work as intended. However, we will revisit testing later on in the course, so extensive testing is not yet recommended. Similarly, your code should have a reasonable number of comments, but documentation is going to be the topic of a future assignment. As a general guideline at this stage of the course, comments and tests should be similar to those accepted in EECS 132. Additionally, comments should only be applied to the code sections that you feel are not self-documenting.

Discussion Guidelines

Although the discussion can range through the whole reading assignment, the emphasis will be on:

- Functions that exceed McCabe's complexity of 4 (if any)
- Non-structured programming constructs and break statements (if any)

Submission

Submit an electronic copy of your program to Canvas. In addition to your code, include a README file explaining how to compile and run the code. The code should be handed in a zip, tar.bz2, or tar.gz archive. Archives in 7z cannot be accepted. You can either bring a laptop to discussion to display your project on a projector, or present your project from the canvas submission.

Grading Guidelines

Advance Warning: starting with Programming Assignment 4, an automatic C (or less) is triggered by any routine with complexity greater than 4 or by any substantially repeated piece of code.