EE461L: Software Implementation and Design Lab
Problem Set 1

Out: October 30, 2017; **Due: November 12, 2017 11:59pm**
Submission: *.zip via Canvas
Maximum points: 50

Generating Java programs using the Alloy API

In this homework, you are to use the Alloy API to solve Alloy constraints that represent the properties of the Java typesystem, enumerate the solutions, and translate them into strings as described below. You need to download the Alloy 4.2 jar (alloy4.2.jar) from: `http://alloy.mit.edu/alloy/download.html`. You can find some examples on how to use the Alloy API here: `http://alloy.mit.edu/alloy/alloy-api-examples.html`. Specifically, the "example using the compiler" is the most relevant to this homework and we use it as a basis for this homework. You can find the Javadoc documentation for Alloy API here: `http://alloy.mit.edu/alloy/documentation/alloy-api/`.

Consider the following Alloy model to partially model the Java type hierarchy:

```
module typehierarchy

abstract sig Type {
  ext: set Type -- ext: Type x Type is a binary relation
}

abstract sig Class extends Type { -- (disjoint) subset
  impl: set Interface -- impl: Class x Interface
}

sig Concrete extends Class {}

sig Abstract extends Class {}

sig Interface extends Type {} -- (disjoint) subset

one sig Object extends Concrete {} -- singleton set
fact { no Object.ext } -- Object.ext is empty set
fact { all c: Class - Object | Object in c.^ext }
fact { no Object.impl } -- Object does not implement interface

-- interface extends interface
fact { all t: Interface | t.ext in Interface }

-- class extends class
fact { all c: Class | c.ext in Class }

-- single inheritance for classes
fact { all c: Class | lone c.ext } -- lone is at most one

-- hierarchy is acyclic
fact { all t: Type | t !in t.^ext } -- ^ is transitive closure

run {} for 2
```

Consider the following skeletal Java code that declares the class `ExampleUsingAlloyAPI`:

```
package pset1;

/*
 * Derived from edu.mit.csail.sdg.alloy4whole.ExampleUsingTheCompiler.java
 * http://alloy.mit.edu/alloy/code/ExampleUsingTheCompiler.java
 *
 */

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import edu.mit.csail.sdg.alloy4.A4Reporter;
import edu.mit.csail.sdg.alloy4.Err;
import edu.mit.csail.sdg.alloy4compiler.ast.Command;
import edu.mit.csail.sdg.alloy4compiler.ast.Module;
import edu.mit.csail.sdg.alloy4compiler.ast.Sig;
import edu.mit.csail.sdg.alloy4compiler.parser.CompUtil;
import edu.mit.csail.sdg.alloy4compiler.translator.A4Options;
import edu.mit.csail.sdg.alloy4compiler.translator.A4Solution;
import edu.mit.csail.sdg.alloy4compiler.translator.A4Tuple;
import edu.mit.csail.sdg.alloy4compiler.translator.A4TupleSet;
import edu.mit.csail.sdg.alloy4compiler.translator.TranslateAlloyToKodkod;

public class ExampleUsingAlloyAPI {
        final static String PATH = "...";

        public static void main(String[] args) throws Err {
                String filename = PATH + "typehierarchy.als";
                A4Reporter rep = new A4Reporter();

                // Parse+typecheck the model
                System.out.println("=========== Parsing+Typechecking "+filename+" =============");
                Module world = CompUtil.parseEverything_fromFile(rep, null, filename);

                // Set options for how to execute the command
                A4Options options = new A4Options();
                options.solver = A4Options.SatSolver.SAT4J;

                Command command = world.getAllCommands().get(0);
                System.out.println("============ Command "+command+": ============");

                // generate and store all solutions
                List<A4Solution> allSols = new ArrayList<A4Solution>();
                int count = findAllSols(rep, world, options, command, allSols);
                System.out.println("number of solutions: " + count);

                // translate each solution into the corresponding Java program
                System.out.println("-----------");
                for (A4Solution sol: allSols) {
                        String program = createProgram(sol,
                                        getRelation(sol, "Type", "ext"),
                                        getRelation(sol, "Class", "impl"));
                        System.out.print(program);
                        System.out.println("-----------");
                }
        }
```

1 Computing all solutions

Complete the implementation of the following method `findAllSols` (in class `ExampleUsingAlloyAPI`) as specified in the comments:

```
        private static int findAllSols(A4Reporter rep, Module world,
                        A4Options options, Command command, List<A4Solution> allSols) throws Err {
                // execute the given command using TranslateAlloyToKodkod.execute_command(...)
                // add each solution found to allSols in order
                // return the number of solutions found
                // hint: study the implementations of
                //   edu.mit.csail.sdg.alloy4whole.ExampleUsingTheCompiler.java and
                //   EvaluatorExample at http://alloy.mit.edu/alloy/alloy-api-examples.html

                // your code goes here

        }
```

To illustrate, running this method on the “typehierarchy.als” model returns 4 as the result and modifies `allSols` to include the following 4 solutions (illustrated as strings):

```
---INSTANCE---
integers={}
univ={Object$0}
Int={}
seq/Int={}
String={}
none={}
this/Object={Object$0}
this/Concrete={Object$0}
this/Abstract={}
this/Class={Object$0}
this/Class<:impl={}
this/Interface={}
this/Type={Object$0}
this/Type<:ext={}

---INSTANCE---
integers={}
univ={Object$0, Interface$0}
Int={}
seq/Int={}
String={}
none={}
this/Object={Object$0}
this/Concrete={Object$0}
this/Abstract={}
this/Class={Object$0}
this/Class<:impl={}
this/Interface={Interface$0}
this/Type={Object$0, Interface$0}
this/Type<:ext={}

---INSTANCE---
integers={}
univ={Object$0, Abstract$0}
Int={}
seq/Int={}
String={}
none={}
this/Object={Object$0}
this/Concrete={Object$0}
this/Abstract={Abstract$0}
```

```
this/Class={Object$0, Abstract$0}
this/Class<:impl={}
this/Interface={}
this/Type={Object$0, Abstract$0}
this/Type<:ext={Abstract$0->Object$0}

---INSTANCE---
integers={}
univ={Object$0, Concrete$0}
Int={}
seq/Int={}
String={}
none={}
this/Object={Object$0}
this/Concrete={Object$0, Concrete$0}
this/Abstract={}
this/Class={Object$0, Concrete$0}
this/Class<:impl={}
this/Interface={}
this/Type={Object$0, Concrete$0}
this/Type<:ext={Concrete$0->Object$0}
```

2 Translating an Alloy field to a Java map

Complete the implementation of the following method `getRelation` (in class `ExampleUsingAlloyAPI`) as specified in the comments:

```
private static Map<String, String> getRelation(A4Solution sol,
                String signame, String fieldname) {
        // iterate over the sigs and their fields in <sol>
        //   to find field <fieldname> in sig <signame>,
        //   create a map that represents the tuples in the
        //   corresponding relation, return the map
        // hint: use methods A4Solution.getAllReachableSigs() and
        //   Sig.getFields() to iterate over all sigs and fields in <sol>;
        //   use method A4Solution.eval(f) to get the value of field f in <sol>;
        //   use method A4Tuple.atom(i) to get atom at position i in the tuple

        // your code goes here

}
```

To illustrate, running this method to translate the field “ext” in sig “Type” for the following Alloy instance:

```
---INSTANCE---
integers={}
univ={Object$0, Concrete$0}
Int={}
seq/Int={}
String={}
none={}
this/Object={Object$0}
this/Concrete={Object$0, Concrete$0}
this/Abstract={}
this/Class={Object$0, Concrete$0}
this/Class<:impl={}
this/Interface={}
this/Type={Object$0, Concrete$0}
this/Type<:ext={Concrete$0->Object$0}
```

returns the following Java map: “`{Concrete$0=Object$0}`”.

3 Translating Alloy instances to corresponding Java programs

Complete the implementation of the following method `createProgram` (in class `ExampleUsingAlloyAPI`) as specified in comments:

```
private static String createProgram(A4Solution sol,
                Map<String, String> supertype,
                Map<String, String> implementS) {
        // assume input map <supertype> is already initialized
        //   to represent the value of "ext" relation in <sol>
        // assume input map <implementS> is already initialized
        //   to represent the value of "impl" relation in <sol>
        // return the Java program represented by <sol>

        // your code goes here

}
```

To illustrate, running this method on the following Alloy instance:

```
---INSTANCE---
integers={}
univ={Object$0, Concrete$0}
Int={}
seq/Int={}
String={}
none={}
this/Object={Object$0}
this/Concrete={Object$0, Concrete$0}
this/Abstract={}
this/Class={Object$0, Concrete$0}
this/Class<:impl={}
this/Interface={}
this/Type={Object$0, Concrete$0}
this/Type<:ext={Concrete$0->Object$0}
```

returns the following Java program: “`class C0 {}`”.

As another illustration, running the `main` method (in class `ExampleUsingAlloyAPI`) outputs the following to the console:

```
=========== Parsing+Typechecking ...
============ Command Run run$1 for 2: ============
number of solutions: 4
-----------
-----------
interface I0 {}
-----------
abstract class A0 {}
-----------
class C0 {}
-----------
```

If we change the scope in the Alloy model to 3, i.e., modify the command in the “typehierarchy.als” model to “`run {} for 3`”, and then run the `main` method, the following console output is produced:

```
=========== Parsing+Typechecking ...
============ Command Run run$1 for 3: ============
number of solutions: 17
-----------
abstract class A0 {}
abstract class A1 {}
-----------
abstract class A0 implements I0 {}
```

```
interface I0 {}
-----------
class C0 implements I0 {}
interface I0 {}
-----------
class C0 {}
interface I0 {}
-----------
abstract class A0 {}
interface I0 {}
-----------
class C0 {}
-----------
abstract class A0 {}
-----------
abstract class A0 extends A1 {}
abstract class A1 {}
-----------
-----------
interface I0 {}
-----------
interface I0 {}
interface I1 extends I0 {}
-----------
interface I0 {}
interface I1 {}
-----------
class C0 {}
abstract class A0 extends C0 {}
-----------
class C0 extends A0 {}
abstract class A0 {}
-----------
class C0 extends C1 {}
class C1 {}
-----------
class C0 {}
class C1 {}
-----------
class C0 {}
abstract class A0 {}
-----------
```

Note how the names of classes and interfaces have been simplified (in comparison to the atoms names in Alloy instances). Your code must also simplify them in the same way.