CMPT 489 / 980 Program Synthesis

Final Project

Phase I is due by 11:59pm PT on Wednesday Nov 8, 2023. Phase II is due by 11:59pm PT on Tuesday Dec 5, 2023. Please submit them to Canvas on time. No late submission is accepted.

Requirements:

- This project must be your own work. No collaboration is permitted.
- The programming language of this project is Java 11.
- You can learn the code on slides and start from it.
- You can use third-party libraries but not existing synthesizers. However, you can implement the algorithms in existing synthesizers by yourself.

1 Problem Description

Consider the following context-free grammar G

$$\begin{array}{rcl} E & ::= & \texttt{Ite}(B, E, E) \mid \texttt{Add}(E, E) \mid \texttt{Multiply}(E, E) \mid x \mid y \mid z \mid 1 \mid 2 \mid 3 \\ B & ::= & \texttt{Lt}(E, E) \mid \texttt{Eq}(E, E) \mid \texttt{And}(B, B) \mid \texttt{Or}(B, B) \mid \texttt{Not}(B) \end{array}$$

$$x, y, z \in \textbf{Variables} \qquad 1, 2, 3 \in \textbf{Constants}$$

Here, E is the start symbol. E and B are non-terminals; all other symbols are terminals. The meaning of terminal symbols are self-explanatory. Specifically, `Ite` is the if-then-else operator. `Add` is the addition (+) operator. `Multiply` is the multiplication (∗) operator. x, y, z are integer variables. $1, 2, 3$ are integer constants. `Lt` is the less-than (<) operator. `Eq` is the equals (==) operator. `And` is the logical conjunction (&&). `Or` is the logical disjunction (||). `Not` is the logical negation (!).

In this project, you need to write an example-based program synthesizer in Java. Specifically, the synthesizer takes as input a list of input-output examples and the context-free grammar G and produces as output an implementation of $f(x, y, z)$ in the language of G such that $f(x, y, z)$ is consistent with the provided examples. You can assume f only uses three variables x, y, z, and all their types are **Int**. The return type of f is also **Int**. If the synthesis succeeds, your program should print the program, e.g., `Add(Add(y, z), x)`, to the console. Otherwise, if the synthesis fails, the program should print `null`.

2 Codebase

A codebase is provided as the starting point. It contains the basic framework for the synthesizer. Details are explained as follows.

Package synth.cfg

This package defines the data structure for the context-free grammar (CFG). It has the following classes

- `Symbol`. An abstract class for symbols in the CFG.
- `Terminal`. A subclass of `Symbol` that corresponds to terminals in the CFG.
- `NonTerminal`. A subclass of `Symbol` that corresponds to non-terminals in the CFG.

- `Production`. A class for productions in the CFG. A production is of the form

  ReturnSymbol ::= Operator(ArgSymbol, ..., ArgSymbol)

- `CFG`. A class for representing the CFG. The most important method is `getProductions`, which takes as input a non-terminal symbol N and returns as output a list of all productions with N being the left-hand-side of the production.

Package synth.core

This package contains the classes for synthesizers, examples, programs, and interpreters.

- `ASTNode`. A class for general Abstract Syntax Tree (AST) nodes. The `symbol` fields corresponds to the symbol in the CFG. The `children` field corresponds to the children nodes.
- `Program`. A class for representing a program. It only has one field `root`, which is the root node of the corresponding AST.
- `Example`. A class that defines the data structure of an example. The `input` field is a map from variable names to their values. The `output` field is the output value.
- `Interpreter`. A class that defines an interpreter of the language of G. The most important method is the static method `evaluate`, which takes as input a program and an environment and returns as output the evaluation result. The environment is essentially a map from variable names to their values, just like the `input` field of `Example`. Concrete examples on how to use `Interpreter.evaluate` are provided in the test class `synth.core.InterpreterTests`.
- `ISynthesizer`. An interface that defines the input and output of a synthesizer. The inputs are a CFG and a list of examples. The output is a program.
- `TopDownEnumSynthesizer`. A top-down enumerative synthesizer that implements the `ISynthesizer` interface. **You need to implement this class.**

Package synth.util

This package contains the utility classes and methods.

- `FileUtils`. A class for file operations. The `readLinesFromFile` static method reads a file into a list of strings, where each line of the file is a string.
- `Parser`. A class for parsing the examples. The `parseAnExample` static method parses text of the form "`x=a, y=b, z=c -> d`" to an object of class `Example`. The `parseAllExamples` static method parses a list of examples from a list of strings, where each string corresponds to an example. It ignores empty strings.

Class synth.Main

The main class of the framework. It has two methods.

- `main`. It is the entry of the program. It takes one command-line argument `args[0]` for the path to the examples file.
- `buildCFG`. It builds the CFG G in Section 1.

Tests

JUnit tests are provided in the `test` directory. You are welcome to add more!

- `synth.core.InterpreterTests`. It contains several unit tests for the interpreter, which is also helpful for understanding the usage of the interpreter.

Other Files

- `pom.xml`. The configuration file for Maven.
- `examples.txt`. A sample examples file.

3 Compilation and Execution

Compilation. This codebase uses the Maven build system. Suppose you enter the `Synth` directory, the project can be easily compiled with one command

```
$ mvn package
```

Then you should be able to see the message "`BUILD SUCCESS`". A directory called `target` will be created and a jar file called `synth-1.0.jar` will be generated inside the `target`.

Execution. In the `Synth` directory, you can execute the program using command

```
$ java -cp lib:target/synth-1.0.jar synth.Main <path-to-examples-file>
```

where `<path-to-examples-file>` is the path to the examples file. For example, you can run

```
$ java -cp lib:target/synth-1.0.jar synth.Main examples.txt
```

You will see a runtime exception with message "`To be implemented`", because the synthesizer is not implemented yet. After you finish implementing the synthesizer, you should see something like (not unique)

```
Add(Add(y, z), x)
```

4 Phase I

In Phase I, you need to implement a top-down enumerative synthesizer in `synth.core.TopDownEnumSynthesizer`.

Deliverable

A zip file called `Phase1_Firstname_Lastname.zip` that contains at least the followings:

- The entire `Synth` directory. You can change existing code if you want, but please make sure the project can be compiled and executed as explained in Section 3.
- A short report (1-2 pages) called `Phase1_Firstname_Lastname.pdf` that explains the design choices, features, tests, issues (if any), and anything else that you want to explain about your program.

5 Phase II

In Phase II, you can implement any synthesis algorithm that improves the performance of the synthesizer on the same problem. **You also need to create a small benchmark set and evaluate your algorithm over the benchmarks.**

A zip file called `Phase2_Firstname_Lastname.zip` that contains at least the followings:

- The entire `Synth` directory. You can change existing code if you want, but please make sure the project can be compiled and executed as explained in Section 3.
- A long report (5-6 pages) called `Phase2_Firstname_Lastname.pdf` that explains the **algorithms**, **benchmarks**, **evaluation results**, design choices, features, tests, issues (if any), and anything else that you want to explain about your program.