Programming Assignment 2: Dead Code Detection

1 Assignment Objectives

- Implement a dead code detector for Java.

Dead code *elimination* is a common compiler optimization in which dead code is removed from a program, and its most challenging part is the *detection* of dead code. In this programming assignment, you will implement a dead code detector for Java by incorporating the analyses you wrote in the previous two assignments, i.e., *live variable analysis* and *constant propagation*. In this document, we will define the scope of dead code (in this assignment) and your task is to implement a detector to recognize them.

2 Introduction to Dead Code Detection

Dead code is the part of program which is unreachable (i.e., never be executed) or is executed but whose results are never used in any other computation. Eliminating dead code does not affect the program outputs, meanwhile, it can shrink program size and improve the performance. In this assignment, we focus on two kinds of dead code, i.e., *unreachable code* and *dead assignment*.

2.1 Unreachable Code

Unreachable code in a program will never be executed. We consider two kinds of unreachable code, *control-flow unreachable code* and *unreachable branch*, as introduced below.

Control-flow Unreachable Code. In a method, if there exists no control-flow path to a piece of code from the entry of the method, then the code is control-flow unreachable. An example is the code following *return statements*. *Return statements* are exits of a method, so the code following them is unreachable. For example, the code at lines 4 and 5 below is control-flow unreachable:

```
1 int controlFlowUnreachable() {
2     int x = 1;
3     return x;
4     int z = 42; // control-flow unreachable code
5     foo(z); // control-flow unreachable code
6 }
```

Detection: such code can be easily detected with control-flow graph (CFG) of the method. We just need to traverse the CFG from the method entry and mark reachable statements. When the traversal finishes, the unmarked statements are control-flow unreachable.

Unreachable Branch. There are two kinds of branching statements in Java: *if* statement and *switch* statement, which could form unreachable branches.

For an if-statement, if its condition value is a constant, then one of its two branches is certainly unreachable in any executions, i.e., unreachable branch, and the code inside that branch is unreachable. For example, in the following code snippet, the condition of if-statement at line 3 is always true, so its false-branch (line 6) is a unreachable branch.

```
1 int unreachableIfBranch() {
2     int a = 1, b = 0, c;
3     if (a > b)
4         c = 2333;
5     else
6         c = 6666; // unreachable branch
7     return c;
8 }
```

For a switch-statement, if its condition value is a constant, then case branches whose values do not match that condition may be unreachable in any execution and become unreachable branches. For example, in the following code snippet, the condition value (`x`) of switch-statement at line 3 is always 2, thus the branches "`case 1`" and "`default`" are unreachable. Note that although branch "`case 3`" does not match the condition value (2) either, it is still reachable as the control flow can fall through to it via branch "`case 2`".

```
1 int unreachableSwitchBranch() {
2     int x = 2, y;
3     switch (x) {
4         case 1: y = 100; break; // unreachable branch
5         case 2: y = 200;
6         case 3: y = 300; break; // fall through
7         default: y = 666; // unreachable branch
8     }
9     return y;
10 }
```

Detection: to detect unreachable branches, we need to perform a constant propagation in advance, which tells us whether the condition values are constants, and then during CFG traversal, we do not enter the corresponding unreachable branches.

2.2 Dead Assignment

A local variable that is assigned a value but is not read by any subsequent instructions is referred to as a dead assignment, and the assigned variable is *dead variable* (opposite to *live variable*). Dead assignments do not affect program outputs, and thus can be eliminated. For example, the code at lines 3 and 5 below are dead assignments.

```
1 int deadAssign() {
2     int a, b, c;
3     a = 0; // dead assignment
4     a = 1;
5     b = a * 2; // dead assignment
6     c = 3;
7     return c;
8 }
```

Detection: to detect dead assignments, we need to perform a live variable analysis in advance. For an assignment, if its LHS variable is a dead variable (i.e., not live), then we could mark it as a dead assignment, except one case as discussed below.

There is a caveat about dead assignment. Sometimes an assignment `x = expr` cannot be removed even `x` is a dead variable, as the RHS expression *`expr`* may have some side-effects. For example, *`expr`* is a method call (`x = m()`) which could have many side-effects. For this issue, we have provided an API for you to check whether the RHS expression of an assignment may have side-effects (described in Section 3.2). If so, you should not report it as dead code for safety, even `x` is not live.

3 Soot Classes You Need to Know

3.1 Soot Classes for Live Variable Analysis

To implement live variable analysis, you need to obtain the variables that are defined and used in a statement. `soot.Unit` provides two convenient APIs for this:

- `List<ValueBox> getDefBoxes()`
- `List<ValueBox> getUseBoxes()`

The defined/used `soot.Value`s are wrapped in list of `ValueBox`s (i.e., return values). You could obtain the defined local variable of a `Unit` like this:

```
List<ValueBox> defBoxes = unit.getDefBoxes();
for (ValueBox vb : defBoxes) {
    Value value = vb.getValue();
    if (value instanceof Local) {
        Local var = (Local) value; // <- defined variable
        …
    }
}
```

The used variables can be obtained in similar way. If you are interested in more details about `Unit`, `Value`, and `ValueBox`, please refer to these documentations:
https://github.com/Sable/soot/wiki/Fundamental-Soot-objects
https://www.sable.mcgill.ca/soot/tutorial/pldi03/tutorial.pdf (pages 34--37)

3.2 Soot Classes for Dead Code Detection

- `soot.Body`
  This class represents Jimple bodies of the methods. A `Body` contains all `Units` in the corresponding method, and you could iterate these `Units` through API `getUnits()`:

```
Body body = …;
for (Unit unit : body.getUnits()) {
    …
}
```

- `soot.toolkits.graph.DirectedGraph<Unit>`
  This interface is used to represent the control-flow graphs of the methods, and their nodes are the `Units` of the methods. This interface is iterable, so you could iterate all nodes (`Units`) over a control-flow graph like this:

```
DirectedGraph<Unit> cfg = …;
for (Unit unit : cfg) {
    …
}
```

  When you traverse a CFG, you could obtain the successors of a `Unit` by API `List<Unit> getSuccsOf(Unit)` like this:

```
for (Unit succ : cfg.getSuccsOf(unit)) {
    …
}
```

- `soot.jimple.IfStmt` (subclass of `Unit`)
  This class represents if-statements in the program.
  - `soot.Value getCondition()`: returns the condition expression. You will need to use constant propagation to compute the value of the expression.
  - `Unit getTarget()`: returns the first `Unit` of its true branch.

  `IfStmt` does not provide API to directly obtain the first `Unit` of its false branch. You could obtain it through the `Body` containing the `IfStmt`:

```
Unit falseBranch = body.getUnits().getSuccOf(ifStmt);
```

- `soot.jimple.AssignStmt` (subclass of `Unit`)
  This class represents assignment (i.e., `x = …;`) in the program. You could obtain its LHS variable through this API `soot.Value getLeftOp()`, for example:

```
AssignStmt assign = …;
Local l = (Local) assign.getLeftOp();
```

4 Your Task [Important!]

In this assignment, your task is to use Soot to analyze the dead code in a given java file as described in 2.1 and 2.2 and output the txt file that marking the dead code. As an example,part of your input file ./test/test0.java is

```
2    int x = 1;
3    return x;
4    int z = 42; // control-flow unreachable code
6 }
```

and this part of your output file ./output/test0.txt is

```
Line 4 : int z = 42;  control-flow unreachable code;
```

Note:We only care about the body of the main function, the function name and the rest of the file do not need to be output.

Further, you need to implement your method in one or more files which must contain DeadCodeDetection.java .

We will execute your jar file like" java -jar DeadCodeDetection.jar . /test". The code generates txt files to the . /output folder corresponds to the file of the same name in. /test.

5 General Requirements

- In this assignment, your only goal is correctness. Efficiency is not your concern.
- **DO NOT** distribute the assignment package to any others.
- Last but not least, do **NOT** plagiarize. The work must be all your own!

6 Submission of Assignment

Your submission should be a zip file, which contains your implementation of

- `source code, including DeadCodeDetection.java`
- `DeadCodeDetection.jar`
- `report.pdf,The report describing the implementation of the algorithm.`

The naming convention your submission is: `<STUDENT_ID>-<NAME>-A2.zip`

Please submit your assignment to `Blackboard`.

7 Grading

The points will be allocated for correctness. We will use your submission to analyze the given test files from the `./test` directory, as well as other tests of our own, and compare your output to that of our solution.