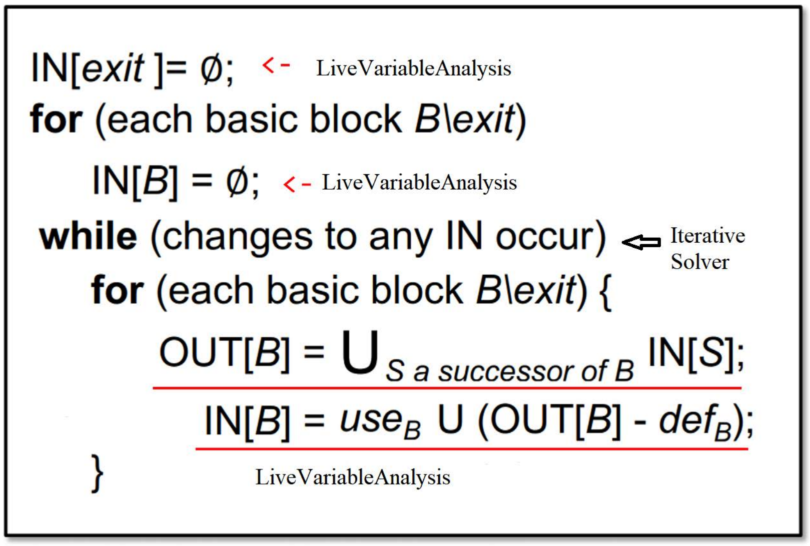
Assignment 1:

Live Variable Analysis and Iterative Solver

1 Assignment Objectives

- Implement a live variable analysis for Java.
- Implement a generic iterative solver, which will be used to solve the data-flow problem you defined, i.e., the live variable analysis.

In this first programming assignment, you will implement a live variable analysis and we divide the algorithm shown in the figure below into two parts Live Variable Analysis ,Iterative Solver .



```
IN[exit] = ∅;   <- LiveVariableAnalysis
for (each basic block B\exit)
    IN[B] = ∅;   <- LiveVariableAnalysis
while (changes to any IN occur)   <= Iterative Solver
    for (each basic block B\exit) {
        $OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S]$;
        $IN[B] = use_B \cup (OUT[B] - def_B)$;
    }          LiveVariableAnalysis
```

Note that in this document (and the documents for other assignments), we only describe the assignment requirements, and to learn the related APIs work, or to get a more comprehensive understanding about the Soot, you need to read the source code and the corresponding document. So please reserve some time for the source code reading and understanding, and this will help improve your ability for rapidly getting familiar with complicated programs.

2 Implementing Live Variable Analysis

2.1 Soot Classes You Need to Know

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.Values` are wrapped in list of `ValueBoxs` (i.e., return values). You could obtain the defined local variable of a `Unit` like this:

```java
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)

2.2 Your Task [Important!]

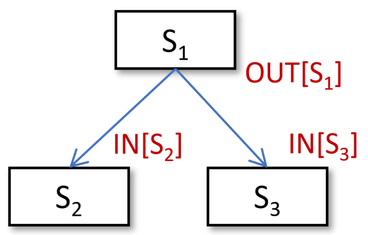
Your first task is to implement the a class named `LiveVariableAnalysis,which achieves the following four parts`:

- `newBoundaryFact(CFG)`
- `newInitialFact()`
- `meetInto(Fact,Fact)`
- `transferNode(Stmt,Fact,Fact)`

which correspond to four parts of live variable analysis algorithm as shown below:

```
IN[$exit$] = $\emptyset$;   <- newBoundaryFact()
for (each basic block $B \setminus exit$)
    IN[$B$] = $\emptyset$;   <- newInitialFact()
while (changes to any IN occur)
    for (each basic block $B \setminus exit$) {
        $\underline{\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]}$;
meetInto()  $\underline{\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)}$;
    }                  transferNode()
```

Here, the design of `meetInto()` might be a bit different from what you expect. It takes two facts as arguments (`fact` and `target`), and is supposed to meet `fact` into `target`. Unlike the equation in the above algorithm which sets OUT[B] to the union of IN facts of all successors of B, `meetInto()` meets the IN fact of each successor to OUT[B] individually, as illustrated by the following example.



Equation: OUT[S_1] = IN[S_2] U IN[S_3]

meetInto(): OUT[S_1] U= IN[S_2] (meetInto(IN[S_2], OUT[S_1]))
OUT[S_1] U= IN[S_3] (meetInto(IN[S_3], OUT[S_1]))

We design `meetInto()` in this way for efficiency. Firstly, each call to `meetInto()` for the same control-flow confluence node manipulates the same `SetFact` object (as OUT fact of the node), thus we can avoid creating many new `Fact` objects (for equation OUT[S] = U…, you may need to create multiple new temporary `Fact` objects when computing the union of IN facts at each iteration for the same node). In addition, such design enables an optimization, i.e., we can avoid meeting unchanged facts. For example, in some iteration, IN[S_2] changes and IN[S_3] remains the same. Then unlike the big union equation in the lecture, we only need to meet IN[S_2] into OUT[S_1] and avoid meeting IN[S_3]. This optimization is beyond the scope of this assignment, and you do not need to consider it in your implementation.

To support such meet strategy, you also need to give OUT[S] an initial fact (the same initial value as in IN[S]) for each statement.

3 Implementing Iterative Solver

Solver holds an object of the corresponding data-flow analysis in its field analysis (LiveVariableAnalysis in this assignment), and you should use it to implement the solver.

3.1 Hints You Need to Know

To implement iterative solver on Soot, we give you some hints. You can implement the corresponding functions through the API provided by Soot according to the description and finally finish this assignment.

- Each object of this part manages the facts of a CFG in a data-flow analysis. You could get/set IN/OUT facts of nodes in a CFG through this part.

- This part represents control-flow graphs of the methods in a program. It is iterable, thus you could iterate the nodes of a CFG via a *for* loop:

```
CFG<Node> cfg = ...;
for (Node node : cfg) {
    ...
}
```

You could iterate predecessors/successors of a node by `CFG.predsOf(Node)` and `CFG.succsOf(Node)`, for example,

```
cfg.succsOf(node).forEach(succ -> {
    ...
});
```

➢ This is the base part for data-flow analysis solvers. In run time, this part of code builds CFGs and passes them to `solver` to start the solver. You can achieve this part through *initialize* and *doSolve* methods, for supporting forward and backward data-flow analyses respectively (despite a bit redundant, such design will lead to a more clean and simpler code structure, and accordingly more straightforward implementation, compared with one analysis for two directions).

4 Your Task [Important!]

In this assignment, your task is to use Soot to analyze a live variable in a given java file and output the txt file that marking the live variable in each line. As an example ,part of a java file ./test/test0.java is

```
    d = a + b;
    b = d;
    c = a;
    return b ;
```

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

```
d = a + b; [a,d]
b = d; [a,b]
c = a; [b]
return b ; []
```

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 LiveVariableAnalysis.java .
We will execute your jar file like" java -jar LiveVariableAnalysis.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 `LiveVariableAnalysis.java`
- `LiveVariableAnalysis.jar`
- `report.pdf,The report describing the implementation of the algorithm.`

The naming convention your submission is: `<STUDENT_ID>-<NAME>-A1.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 as well as other tests of our own, and compare your output to that of our solution.

Good luck!