CS201/F23 Project Phase 2

Ravan Nazaraliyev 862393741

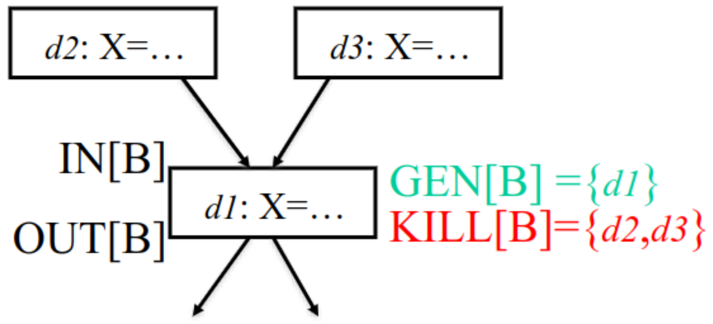
In this part of the project, we analyze the reaching definitions in the given code snippet. The reaching definitions analysis is done according to the material given in the lecture notes.

Let's revise the material:

$$IN[B] = \bigcup_{p \in pred(B)} OUT(p)$$

$$OUT(B) = GEN(B) \cup (IN(B) - KILL(B))$$

This is how we find out INs and OUTs for the targeted basic block B. We also need to determine GEN and KILL for each block.



In the lecture notes, GEN and KILL are calculated in each block. Specifically, GENs are local to each block, but KILL should also take previous definitions of the same variables into consideration since previously defined variables will be KILLed in the current block if they are redefined in the current block. For example, consider the above picture where d1 redefines (store instruction) X, which is a GEN, but since X has been defined previously by the definitions d2 and d3, they are now KILLs.

In my analysis, I consider each store instruction as a new GEN in a given block. However, if the same variable is redefined in the same block, then the previous definition will be removed from GENs, and added to the KILLs.

For example, take the example below:

```
1) int x =  1; <-- store to x
2) int y = 2; <-- store to y
3) int x = 3;  <-- store to x
```

The algorithm takes 1) as a GEN, then 2) as a GEN, then when encountering 3) it sees x is redefined and removes 1) from GEN. Then the algorithm adds 3) to GEN.

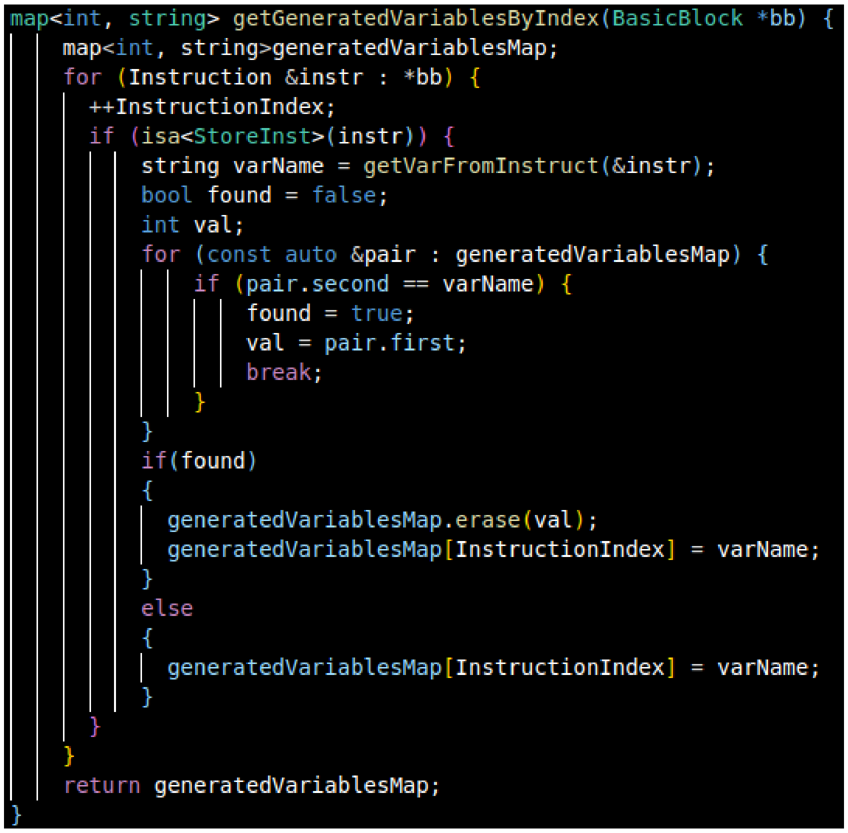
Similarly, for the KILL, the algorithm goes through the instructions and look for the redefinition inside the same block. Different from GEN, KILL determining algorithm also checks the GEN from predecessor blocks. If GEN from pred contains a definition that is redefined in the considered block, then is it a KILL now. Consider the below example:

Considered block:

```
4) x = x + 1; <-- store to x
```

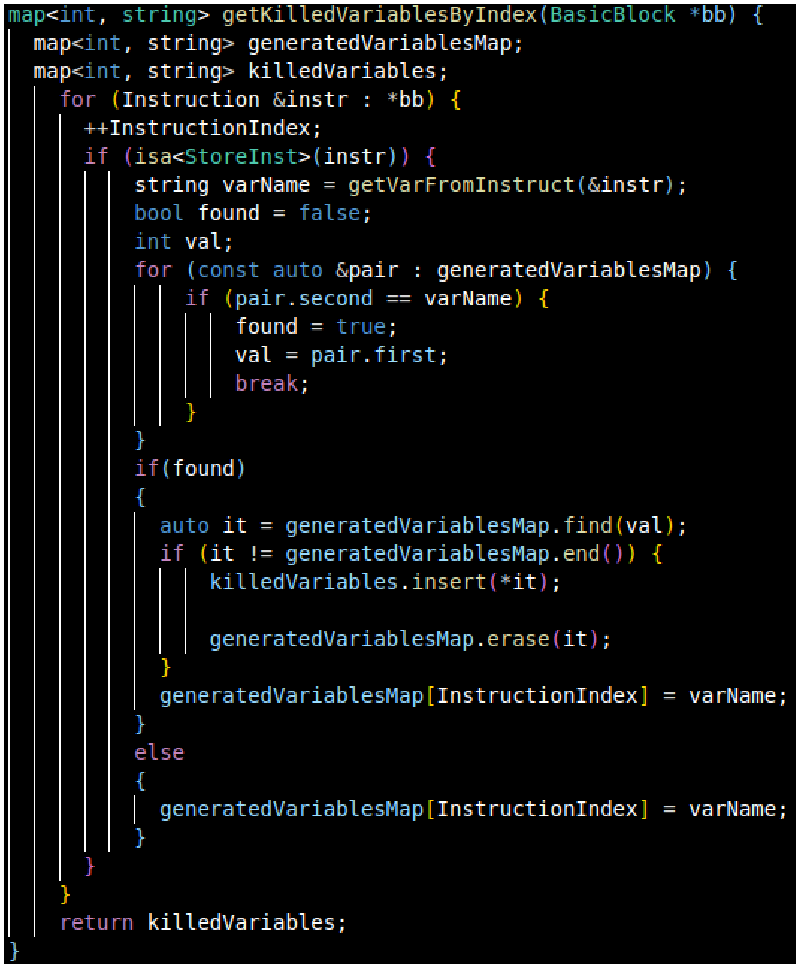
Since x is redefined in this block, 3) is now KILLed.

GEN algorithm:



```
map<int, string> getGeneratedVariablesByIndex(BasicBlock *bb) {
    map<int, string>generatedVariablesMap;
    for (Instruction &instr : *bb) {
      ++InstructionIndex;
      if (isa<StoreInst>(instr)) {
        string varName = getVarFromInstruct(&instr);
        bool found = false;
        int val;
        for (const auto &pair : generatedVariablesMap) {
          if (pair.second == varName) {
            found = true;
            val = pair.first;
            break;
          }
        }
        if(found)
        {
          generatedVariablesMap.erase(val);
          generatedVariablesMap[InstructionIndex] = varName;
        }
        else
        {
          generatedVariablesMap[InstructionIndex] = varName;
        }
      }
    }
    return generatedVariablesMap;
}
```

KILL algorithm:



```
map<int, string> getKilledVariablesByIndex(BasicBlock *bb) {
  map<int, string> generatedVariablesMap;
  map<int, string> killedVariables;
    for (Instruction &instr : *bb) {
      ++InstructionIndex;
      if (isa<StoreInst>(instr)) {
          string varName = getVarFromInstruct(&instr);
          bool found = false;
          int val;
          for (const auto &pair : generatedVariablesMap) {
              if (pair.second == varName) {
                  found = true;
                  val = pair.first;
                  break;
              }
          }
          if(found)
          {
            auto it = generatedVariablesMap.find(val);
            if (it != generatedVariablesMap.end()) {
                killedVariables.insert(*it);

                generatedVariablesMap.erase(it);
            }
            generatedVariablesMap[InstructionIndex] = varName;
          }
          else
          {
            generatedVariablesMap[InstructionIndex] = varName;
          }
      }
    }
    return killedVariables;
}
```

In the algorithm, I use mappings which consists of tuples of IR index and variable since IR index alone will not help in the analysis.

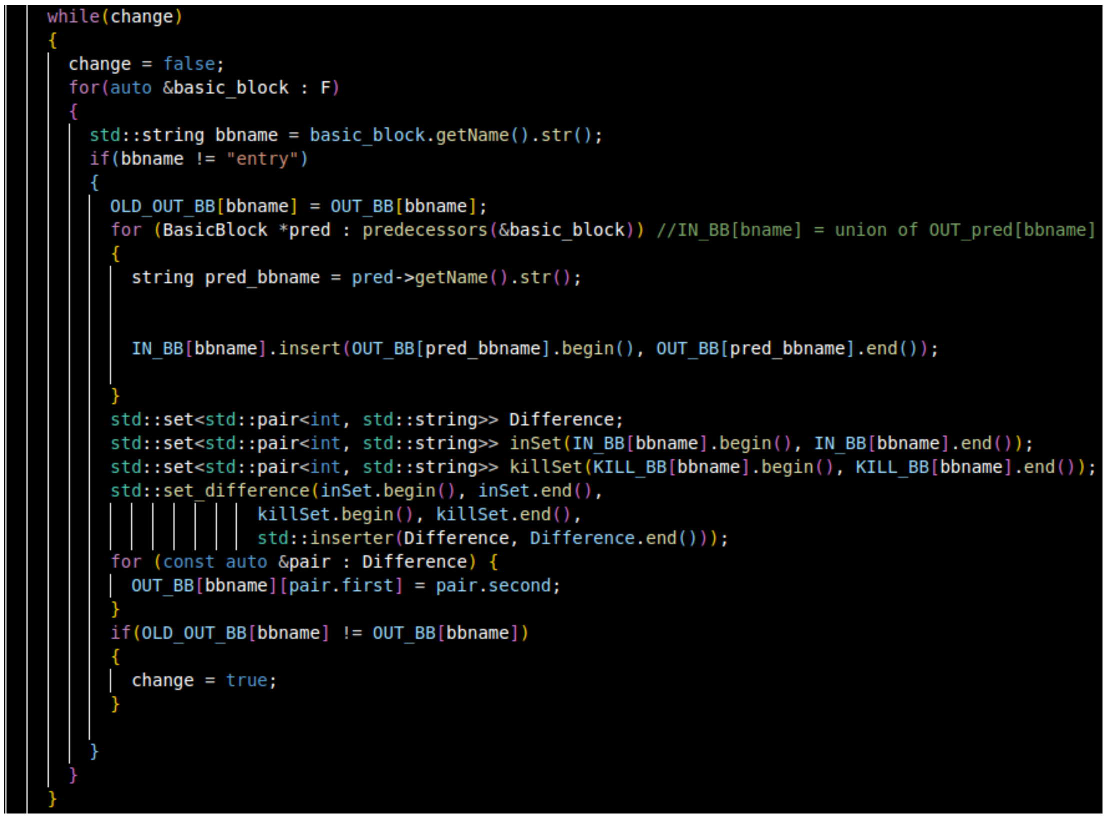
The algorithm is composed of two phases:

1) Preliminary result:
    a. GEN
    b. KILL
    c. OUT = GEN initialization
2) Final result:
    a. Iterative algorithm
    b. Prints all GEN, KILL, IN and OUT

The iterative algorithm is according to the one given in the lecture notes:

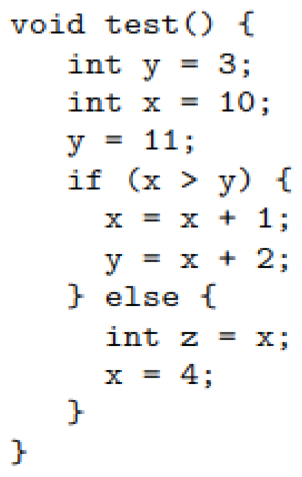
```
-- Initialize sets:
for every block B
   OUT[B] = GEN[B]
   IN[B] = empty list
   --Iteratively solve equations:
   change = true
   while change{
      change = false
      for each B != Be{
         OLD_OUT = OUT[B]
         IN[B] = union of IN[P] where p is in pred of B
         OUT[B] = GEN[B] union (IN[B] - KILL[B])
         if OUT[B] != OLD_OUT
            change = true
      }
   }
```

The reaching definition algorithm is not given in the lecture notes. I adapted the liveness algorithm which is similar to reaching definitions. The only difference in reaching is forward problem. The coding of the worklist algorithm:

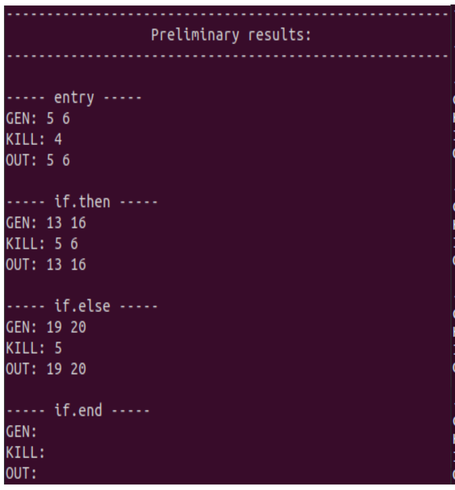


```
while(change)
{
  change = false;
  for(auto &basic_block : F)
  {
    std::string bbname = basic_block.getName().str();
    if(bbname != "entry")
    {
      OLD_OUT_BB[bbname] = OUT_BB[bbname];
      for (BasicBlock *pred : predecessors(&basic_block)) //IN_BB[bname] = union of OUT_pred[bbname]
      {
        string pred_bbname = pred->getName().str();


        IN_BB[bbname].insert(OUT_BB[pred_bbname].begin(), OUT_BB[pred_bbname].end());

      }
      std::set<std::pair<int, std::string>> Difference;
      std::set<std::pair<int, std::string>> inSet(IN_BB[bbname].begin(), IN_BB[bbname].end());
      std::set<std::pair<int, std::string>> killSet(KILL_BB[bbname].begin(), KILL_BB[bbname].end());
      std::set_difference(inSet.begin(), inSet.end(),
                          killSet.begin(), killSet.end(),
                          std::inserter(Difference, Difference.end()));
      for (const auto &pair : Difference) {
        OUT_BB[bbname][pair.first] = pair.second;
      }
      if(OLD_OUT_BB[bbname] != OUT_BB[bbname])
      {
        change = true;
      }

    }
  }
}
```

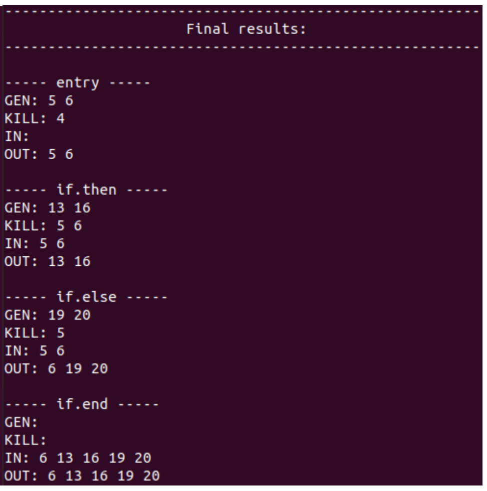
So combining the whole parts, the program is giving the following output for the test codes:



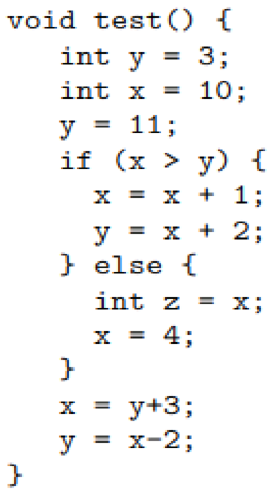
```
void test() {
   int y = 3;
   int x = 10;
   y = 11;
   if (x > y) {
      x = x + 1;
      y = x + 2;
   } else {
      int z = x;
      x = 4;
   }
}
```



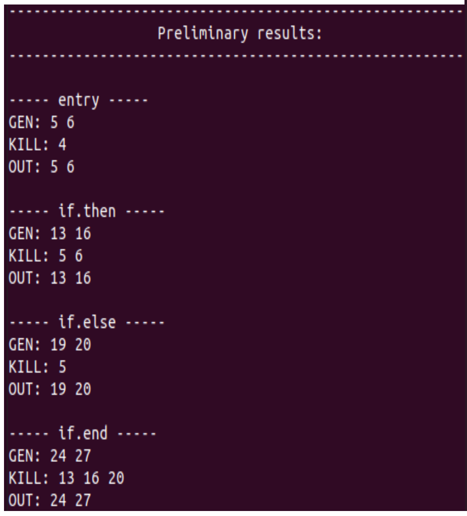
```
------------------------------------------------------
                Preliminary results:
------------------------------------------------------

----- entry -----
GEN: 5 6
KILL: 4
OUT: 5 6

----- if.then -----
GEN: 13 16
KILL: 5 6
OUT: 13 16

----- if.else -----
GEN: 19 20
KILL: 5
OUT: 19 20

----- if.end -----
GEN:
KILL:
OUT:
```



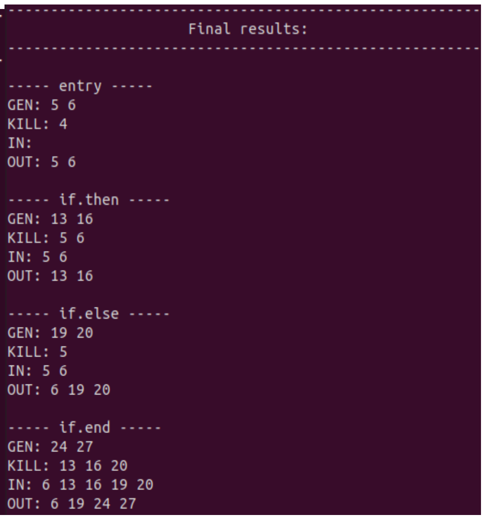
```
------------------------------------------------------
                  Final results:
------------------------------------------------------

----- entry -----
GEN: 5 6
KILL: 4
IN:
OUT: 5 6

----- if.then -----
GEN: 13 16
KILL: 5 6
IN: 5 6
OUT: 13 16

----- if.else -----
GEN: 19 20
KILL: 5
IN: 5 6
OUT: 6 19 20

----- if.end -----
GEN:
KILL:
IN: 6 13 16 19 20
OUT: 6 13 16 19 20
```



```
void test() {
  int y = 3;
  int x = 10;
  y = 11;
  if (x > y) {
    x = x + 1;
    y = x + 2;
  } else {
    int z = x;
    x = 4;
  }
  x = y+3;
  y = x-2;
}
```



```
---------------------------------------------------------
                  Preliminary results:
---------------------------------------------------------

----- entry -----
GEN: 5 6
KILL: 4
OUT: 5 6

----- if.then -----
GEN: 13 16
KILL: 5 6
OUT: 13 16

----- if.else -----
GEN: 19 20
KILL: 5
OUT: 19 20

----- if.end -----
GEN: 24 27
KILL: 13 16 20
OUT: 24 27
```



```
---------------------------------------------------------
                     Final results:
---------------------------------------------------------

----- entry -----
GEN: 5 6
KILL: 4
IN:
OUT: 5 6

----- if.then -----
GEN: 13 16
KILL: 5 6
IN: 5 6
OUT: 13 16

----- if.else -----
GEN: 19 20
KILL: 5
IN: 5 6
OUT: 6 19 20

----- if.end -----
GEN: 24 27
KILL: 13 16 20
IN: 6 13 16 19 20
OUT: 6 19 24 27
```

Only final results should be considered.

In order to run the project:

1) Produce test.ll file:
   a) clang -emit-llvm test.c -S -o test.ll -fno-discard-value-names
2) Build the reachingdefinition c++ file:
   a) In the Pass folder:
   b) mkdir build
   c) cd build
   d) Make
3) To run reaching definition code on given test file:
   a) opt -load Path_to_ ibReachingDefinition.so/libReachingDefinition.so -ReachingDefinition test.ll