HelloWorld.cpp is in llvm-tutor-main\HelloWorld
The test files and the shell script files are in llvm-tutor-main\build

---

Summarize Algorithm and Major Data Structures

Algorithm: Iterative Liveness Analysis

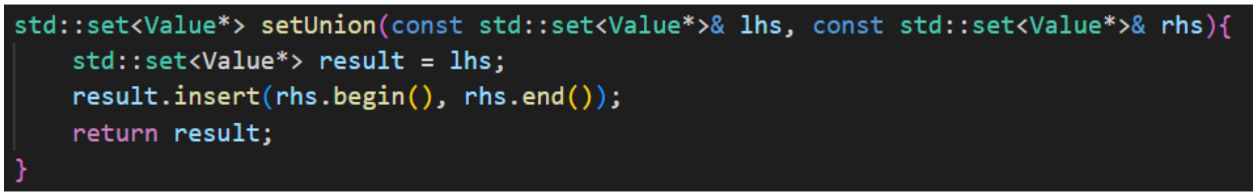
1) **Calculate VarKill and UEVar sets for each basic block in CFG - Initialize both as empty sets.**
    a) **VarKill - For all variables that are redefined, add them to this set.**
    e.g. a ⇜ b + c, VarKill(N) = VarKill(N) ∪ a
    b) **UEVar - Starting from the beginning of the basic block, add variables that are used as operands before they are defined in the block. The intermediate VarKill set for this block can be used to represent variables that are redefined in the block as long as UEVar is modified before VarKill for every particular operation.**
    e.g. a ⇜ b + c, UEVar(N) = UEVar(N) ∪ {b, c}
2) **Calculate LiveOut sets for every basic block in CFG using a basic iterative algorithm.**
    a) **Initialize all LiveOut sets for every basic block as the empty set.**
    b) **Iterate all blocks in CFG continuously to calculate LiveOut sets using the following formula:**
    $$\text{LiveOut}(N) = \cup_{x \in succ(N)} (\text{LiveOut}(X) - \text{VarKill}(X) \cup \text{UEVar}(X))$$
    c) **Stop (b) process when all LiveOut sets don't change from the previous iteration.**

Major Data Structures: Sets and Maps

C++ Maps are utilized to store LiveOut, VarKill, and UEVar sets. The keys are Value pointers representing basic blocks, and the values are the Value pointer sets for the variables in those 3 sets. The sets have union and difference operations performed on them for the LiveOut calculation.

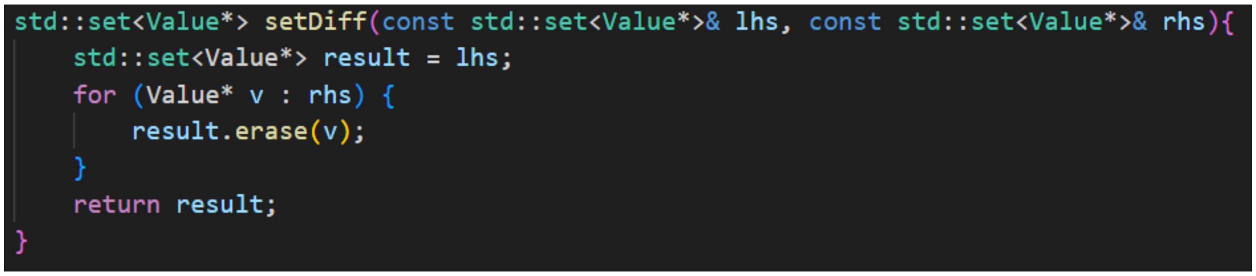
Code Implementation Details

Set Union function used in LiveOut Calculation



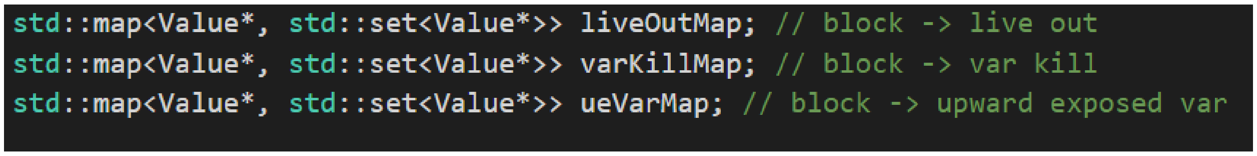
```
std::set<Value*> setUnion(const std::set<Value*>& lhs, const std::set<Value*>& rhs){
    std::set<Value*> result = lhs;
    result.insert(rhs.begin(), rhs.end());
    return result;
}
```

Set Difference function used in LiveOut Calculation



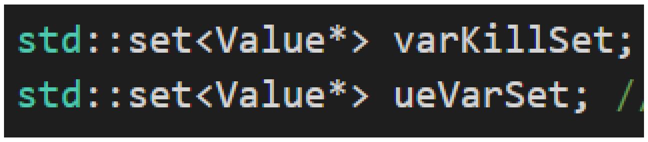
```
std::set<Value*> setDiff(const std::set<Value*>& lhs, const std::set<Value*>& rhs){
    std::set<Value*> result = lhs;
    for (Value* v : rhs) {
        result.erase(v);
    }
    return result;
}
```

Maps for Basic Block → Variable Sets for LiveOut, VarKill, and UEVar



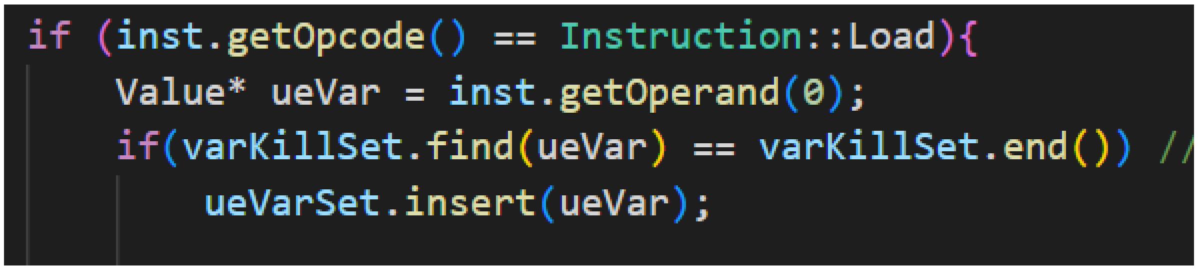
```
std::map<Value*, std::set<Value*>> liveOutMap; // block -> live out
std::map<Value*, std::set<Value*>> varKillMap; // block -> var kill
std::map<Value*, std::set<Value*>> ueVarMap; // block -> upward exposed var
```

For each basic block, fill these intermediate sets for VarKill and UEVar.



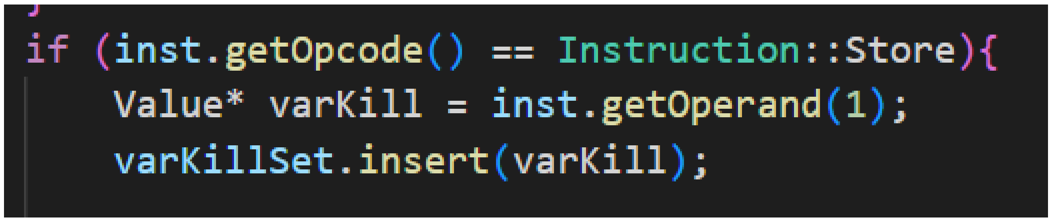
```
std::set<Value*> varKillSet;
std::set<Value*> ueVarSet; /
```

For each load instruction (used as operand), as long as it's not in the intermediate VarKill set, add the source variable to the intermediate UEVar set.



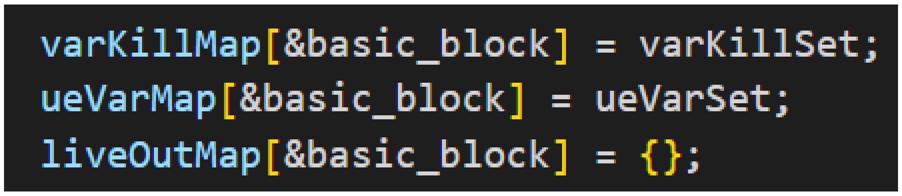
```
if (inst.getOpcode() == Instruction::Load){
    Value* ueVar = inst.getOperand(0);
    if(varKillSet.find(ueVar) == varKillSet.end()) //
        ueVarSet.insert(ueVar);
```

For each store instruction (used as redefinitions), add the destination variable to the intermediate VarKill set.



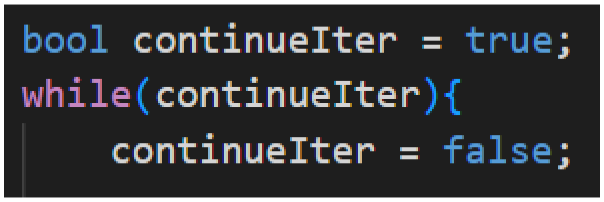
```
if (inst.getOpcode() == Instruction::Store){
    Value* varKill = inst.getOperand(1);
    varKillSet.insert(varKill);
```

Once the full basic block has been calculated, assign the respective sets to the appropriate maps, indexed by basic block pointer. The LiveOut value for each basic block is initialized as the empty set.



```
varKillMap[&basic_block] = varKillSet;
ueVarMap[&basic_block] = ueVarSet;
liveOutMap[&basic_block] = {};
```

Calculating LiveOut, using basic Iterative algorithm, initialize the continue condition.

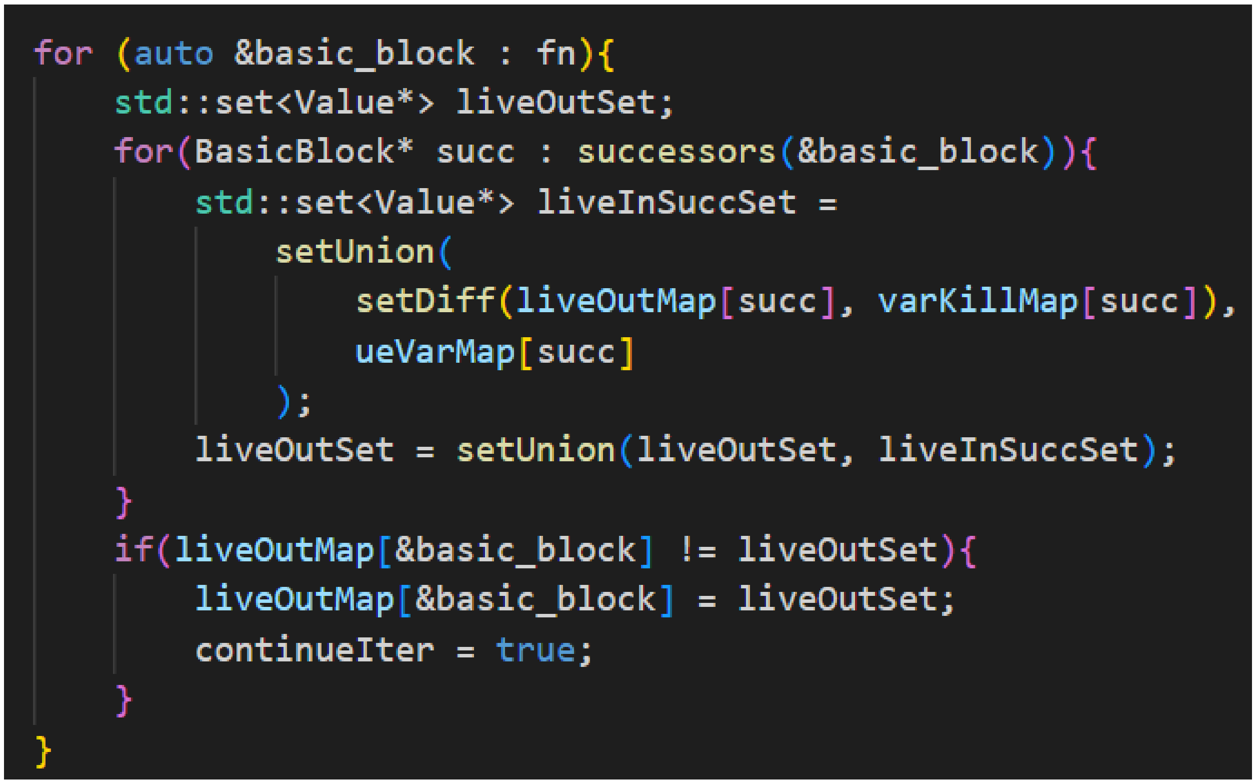


```
bool continueIter = true;
while(continueIter){
    continueIter = false;
```

Loop through all basic blocks in CFG, loop through all successors of the block to calculate the formula for LiveOut:

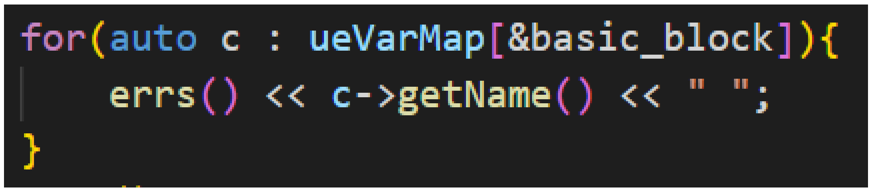
$$\text{LiveOut}(N) = \bigcup_{x \in succ(N)} (\text{LiveOut}(X) - \text{VarKill}(X) \cup \text{UEVar}(X))$$

Utilize the setUnion and setDiff function to perform set operations for the formula. While the successors of the basic block are being processed, perform set union with the liveOutSet and the particular successor's LiveOut set.
As long as at least one block's LiveOut set is modified in an iteration, redo a full iteration of all basic blocks again; this logic is reflected in the if statement.

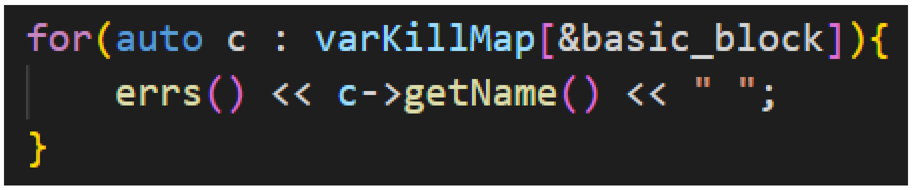


```
for (auto &basic_block : fn){
    std::set<Value*> liveOutSet;
    for(BasicBlock* succ : successors(&basic_block)){
        std::set<Value*> liveInSuccSet =
            setUnion(
                setDiff(liveOutMap[succ], varKillMap[succ]),
                ueVarMap[succ]
            );
        liveOutSet = setUnion(liveOutSet, liveInSuccSet);
    }
    if(liveOutMap[&basic_block] != liveOutSet){
        liveOutMap[&basic_block] = liveOutSet;
        continueIter = true;
    }
}
```

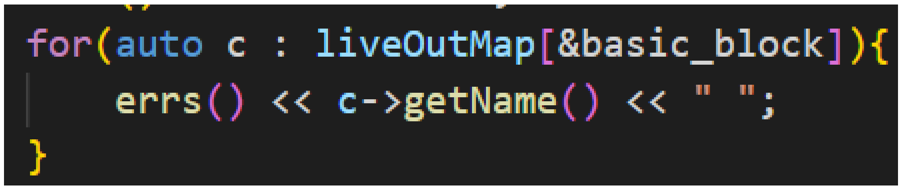
Printing process for output checking, uses the maps, indexed by basic block, and the getName() function to print out all variables in each set.



```
for(auto c : ueVarMap[&basic_block]){
    errs() << c->getName() << " ";
}
```



```
for(auto c : varKillMap[&basic_block]){
    errs() << c->getName() << " ";
}
```



```
for(auto c : liveOutMap[&basic_block]){
    errs() << c->getName() << " ";
}
```