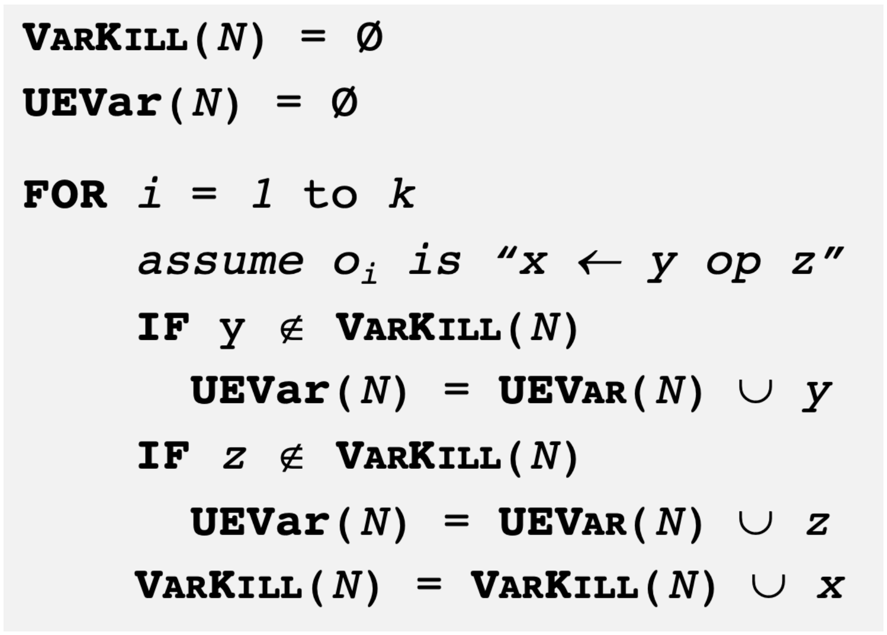
Trung Lam - 861270734
CS201 - Project 3
1 March, 2022

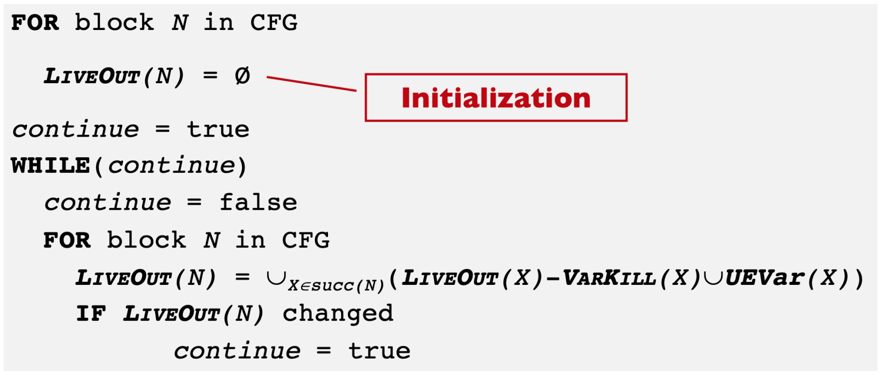
Project 3: Liveness Analysis

Algorithm / Major Data Structure:
Throughout the entire program, the data structure mainly used are maps and vectors. Vectors are used to store the variables of UEVar, VarKill, and LiveOut of a basic block, and maps are used to store the vectors according to the name of the basic blocks. The algorithm used to compute the UEVar and VarKill vectors are similar to the algorithm shown in the lecture slide:



```
VARKILL(N) = $\emptyset$
UEVar(N) = $\emptyset$

FOR i = 1 to k
    assume $o_i$ is "x ← y op z"
    IF y $\notin$ VARKILL(N)
      UEVar(N) = UEVAR(N) $\cup$ y
    IF z $\notin$ VARKILL(N)
      UEVar(N) = UEVAR(N) $\cup$ z
    VARKILL(N) = VARKILL(N) $\cup$ x
```

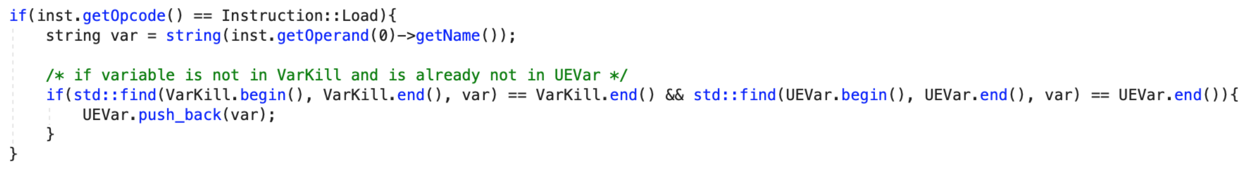
Likewise, the algorithm used to compute LiveOut of each basic block is also similar to the algorithm shown in the lecture slide:


Initialization

```
FOR block N in CFG
  LIVEOUT(N) = $\emptyset$
continue = true
WHILE(continue)
  continue = false
  FOR block N in CFG
    LIVEOUT(N) = $\cup_{X \in succ(N)}$(LIVEOUT(X)−VARKILL(X)$\cup$UEVar(X))
    IF LIVEOUT(N) changed
        continue = true
```

UEVar and VarKill set variable retrieval implementation:

- Load Instructions:



```
if(inst.getOpcode() == Instruction::Load){
    string var = string(inst.getOperand(0)->getName());

    /* if variable is not in VarKill and is already not in UEVar */
    if(std::find(VarKill.begin(), VarKill.end(), var) == VarKill.end() && std::find(UEVar.begin(), UEVar.end(), var) == UEVar.end()){
        UEVar.push_back(var);
    }
}
```

Description: For load instructions, the only thing needed to be concerned is the UEVar set. So based on the algorithm described in the lecture, if it is not in the VarKill set, then add it to the UEVar set. This implementation is similar to the algorithm with the exception of checking whether it is already in the UEVar set. If it is, then we don’t add it to the UEVar vector, otherwise we do.

- Store Instructions:

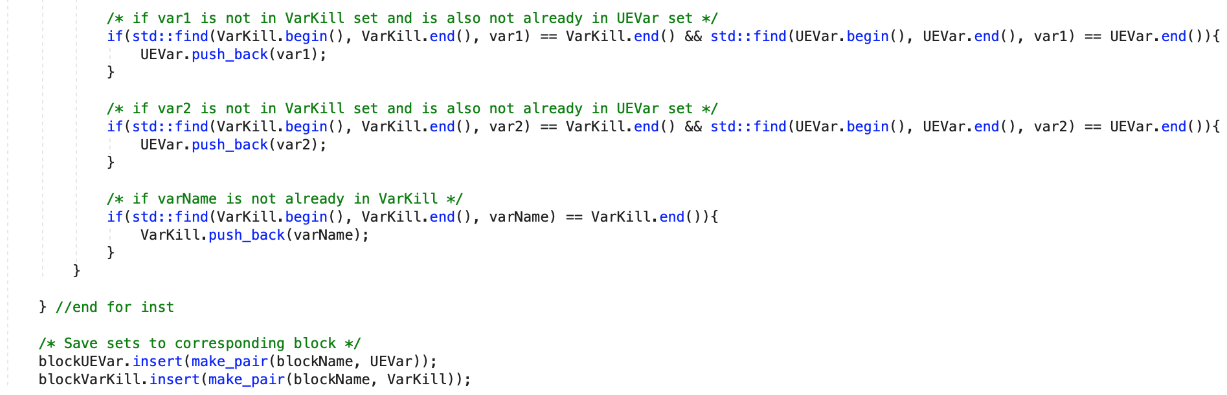
The implementation based on the store instruction in terms of retrieving the variable's name itself depends on a few if statements:

```
if(inst.getOpcode() == Instruction::Store){

    string var1 = "";
    string var2 = "";

    /* if first operand is a constant, ignore */
    if(isa<ConstantInt>(inst.getOperand(0))){
        var1 = "";
    }

    else{
        /* if first operand is a binary op */
        var1 = string(dyn_cast<User>(dyn_cast<User>(inst.getOperand(0))->getOperand(0))->getOperand(0)->getName());
        var2 = string(dyn_cast<User>(dyn_cast<User>(inst.getOperand(0))->getOperand(1))->getOperand(0)->getName());

        /* if first operand returns empty (implies a register or load instruction) */
        if(string(inst.getOperand(0)->getName()) == ""){
            var1 = string(dyn_cast<User>(inst.getOperand(0))->getOperand(0)->getName());
        }

        /* if var1 is a constant */
        if(isa<ConstantInt>(dyn_cast<User>(dyn_cast<User>(inst.getOperand(0))->getOperand(0))->getOperand(0))){
            var1 = "";
        }

        /* if var2 is a constant */
        if(isa<ConstantInt>(dyn_cast<User>(dyn_cast<User>(inst.getOperand(0))->getOperand(1))->getOperand(0))){
            var2 = "";
        }
    }

    /* retrieve name of destination */
    string varName = string(inst.getOperand(1)->getName());
```

Description: For the operands to be used in the UEVar set: if the operand is a constant value, set the name of the variable to an empty string. This includes the operands retrieved from the binary ops. Otherwise, backtrace the registers in the store instruction to retrieve the operand names for the variables. For the operands to be used in the VarKill set, the process of retrieving the name of the variable is simple such that it is only one line of code.

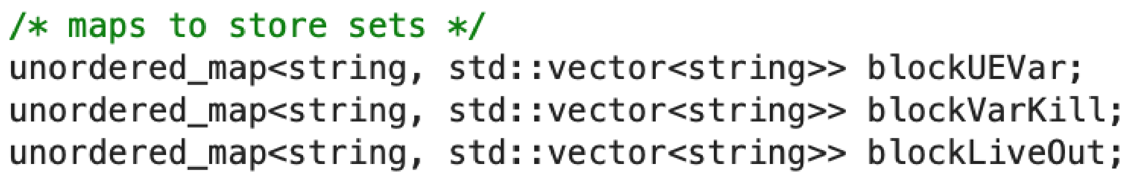
```
/* retrieve name of destination */
string varName = string(inst.getOperand(1)->getName());
```

Computing UEVar and VarKill set implementation:



```
            /* if var1 is not in VarKill set and is also not already in UEVar set */
            if(std::find(VarKill.begin(), VarKill.end(), var1) == VarKill.end() && std::find(UEVar.begin(), UEVar.end(), var1) == UEVar.end()){
                UEVar.push_back(var1);
            }

            /* if var2 is not in VarKill set and is also not already in UEVar set */
            if(std::find(VarKill.begin(), VarKill.end(), var2) == VarKill.end() && std::find(UEVar.begin(), UEVar.end(), var2) == UEVar.end()){
                UEVar.push_back(var2);
            }

            /* if varName is not already in VarKill */
            if(std::find(VarKill.begin(), VarKill.end(), varName) == VarKill.end()){
                VarKill.push_back(varName);
            }
        }

    } //end for inst

    /* Save sets to corresponding block */
    blockUEVar.insert(make_pair(blockName, UEVar));
    blockVarKill.insert(make_pair(blockName, VarKill));
```

Description: The implementation for computing UEVar and VarKill is similar to the algorithm introduced in the lecture. For UEVar, if the operand is not in VarKill, then add to the UEVar vector. To prevent duplicates in the vector, we also check to see whether the operand variable is or is not in the UEVar vector. For the VarKill, the only thing to check, to prevent duplicates, is whether the variable is or is not in the VarKill vector. After getting the UEVar and VarKill of the basic block, save it to the corresponding map of UEVar and VarKill where the key is the block name and the value is the vector of the variables.



```
/* maps to store sets */
unordered_map<string, std::vector<string>> blockUEVar;
unordered_map<string, std::vector<string>> blockVarKill;
unordered_map<string, std::vector<string>> blockLiveOut;
```

Computing LiveOut:

```
/* initialize empty set for LiveOut */
for (auto& basic_block : F){
    vector<string> emptySet = {};
    blockLiveOut.insert(make_pair(string(basic_block.getName()), emptySet));
}

/* Computation of LiveOut */
unordered_map<string ,std::vector<string>>::const_iterator blockLookUp;
bool cont = true;
while(cont){
    cont = false;
    for (auto& basic_block : F){
        std::vector<string> liveOut;
        std::vector<string> liveOutTemp;
        std::vector<string> liveOutSucc;
        std::vector<string> varKillSucc;
        std::vector<string> ueVarSucc;
        std::vector<string> unionSuccessor;

        for(BasicBlock *Succ : successors(&basic_block)){
            std::vector<string> diffTemp;
            std::vector<string> unionTemp;

            /* retrieve current sets of current block */
            blockLookUp = blockLiveOut.find(string(Succ->getName()));
            liveOutSucc = blockLookUp->second;
            blockLookUp = blockVarKill.find(string(Succ->getName()));
            varKillSucc = blockLookUp->second;
            blockLookUp = blockUEVar.find(string(Succ->getName()));
            ueVarSucc = blockLookUp->second;

            /* LiveOut(X) - VarKill(X) */
            std::set_difference(liveOutSucc.begin(), liveOutSucc.end(), varKillSucc.begin(), varKillSucc.end(), std::back_inserter(diffTemp));

            /* U UEVar(X) */
            std::set_union(diffTemp.begin(), diffTemp.end(), ueVarSucc.begin(), ueVarSucc.end(), std::back_inserter(unionTemp));

            for(auto it: unionTemp){
                unionSuccessor.push_back(it);
            }

        }
```

Description: The algorithm for computing LiveOut in terms of structure is similar to the one introduced in lecture. To start, for each basic block, we insert an entry based on the basic block's name and an empty vector to the map blockLiveOut, which stores all LiveOut of all blocks. After which, we run the algorithm. To retrieve the information for the successors, using an iterator, we retrieve the LiveOut, VarKill, and UEVar of the successors from the corresponding maps. After which we use to compute the equation for each successor and store each elements of the result in the unionSuccessor vector:

$$(\mathrm{LiveOut}(X) - \mathrm{VarKill}(X) \cup \mathrm{UEVar}(X))$$

This unionSuccessor vector would contain all elements of the resulting equation for every successor of the basic block. After which, we remove any duplicates of the unionSuccessor vector, giving us the resulting LiveOut of the basic block:

```
/* Compute Union of Successors */
sort(unionSuccessor.begin(), unionSuccessor.end());
unionSuccessor.erase(std::unique(unionSuccessor.begin(), unionSuccessor.end()), unionSuccessor.end());
```

The final step is to check whether LiveOut of that basic block has changed:

```
/* retrieve current LiveOut */
blockLookUp = blockLiveOut.find(string(basic_block.getName()));
liveOut = blockLookUp->second;

/* If LiveOut(N) is changed */
if(liveOut != unionSuccessor){
    cont = true;
}

/* Update LiveOut */
auto it = blockLiveOut.find(string(basic_block.getName()));
it->second = unionSuccessor;
```

First we retrieve the current LiveOut from the blockLiveOut map. After which we compare to the unionSuccessor, which is the new LiveOut. If they are the same, then nothing has changed. If they are different, then we set the bool value of cont to be true. We then update the LiveOut value for that basic block for future computations in the blockLiveOut map.