Assignment 2

Flávio Cruz and Richard Veras

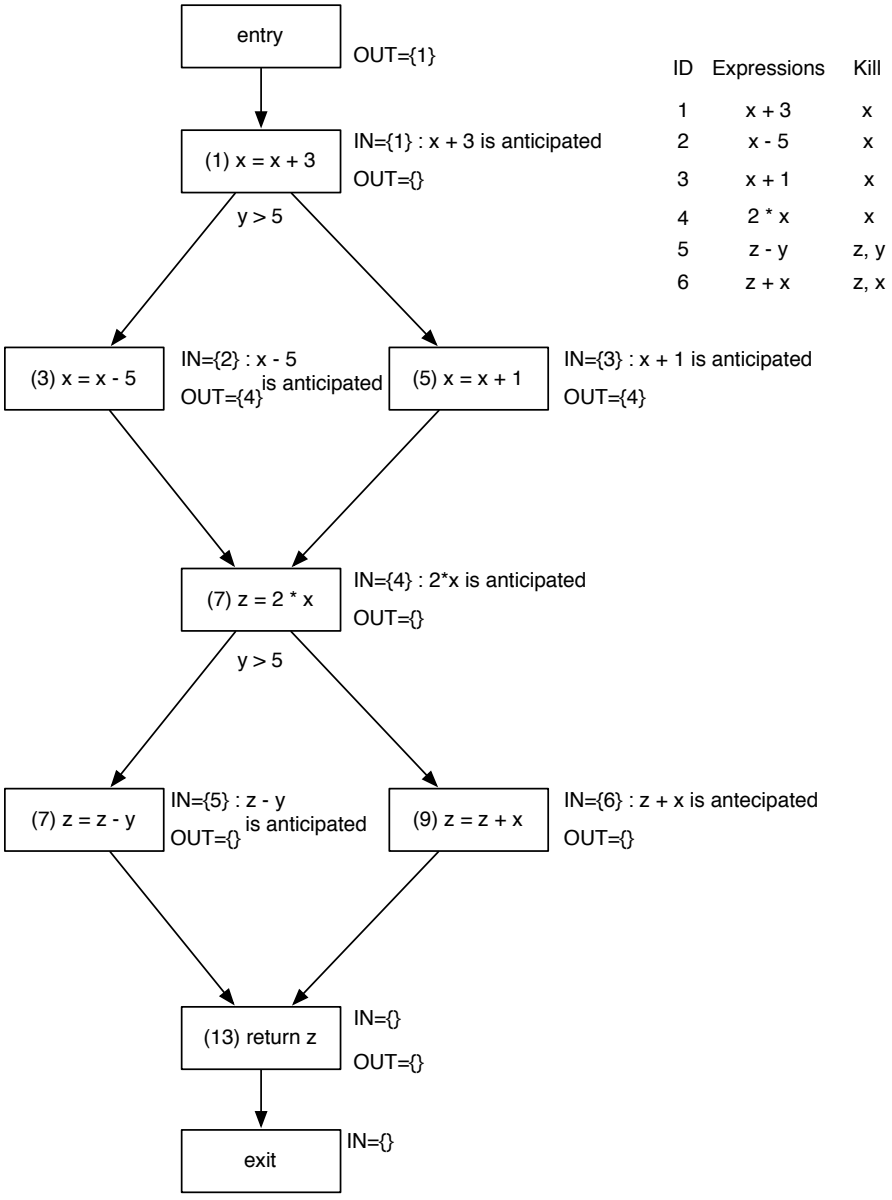
February 16, 2012

1 Questions

1.1 Lazy Code Motion

Item 1

Figure 1: CFG after pass 1: anticipated expressions.



Item 2

Figure 2: CFG after pass 2: early placement.

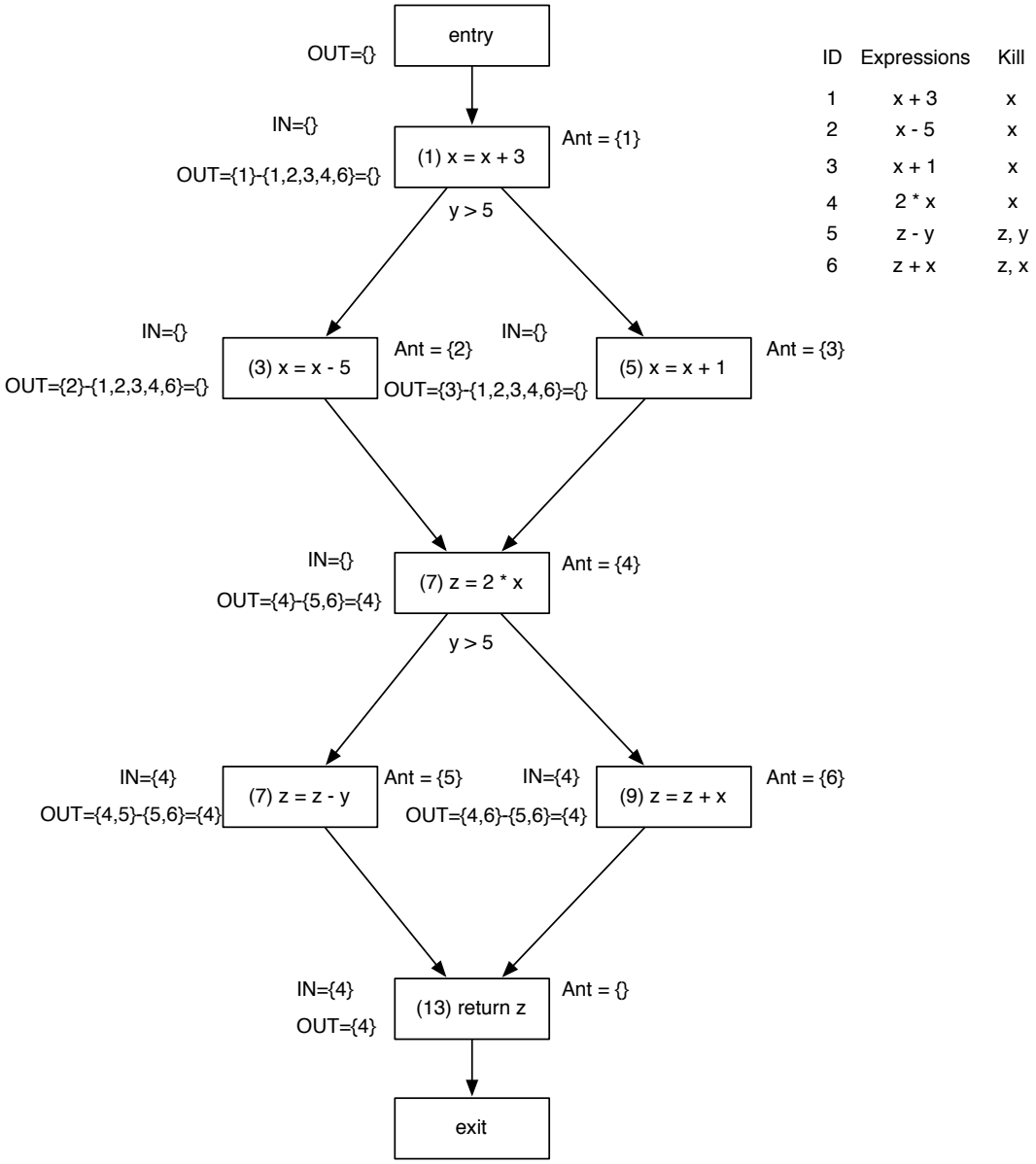
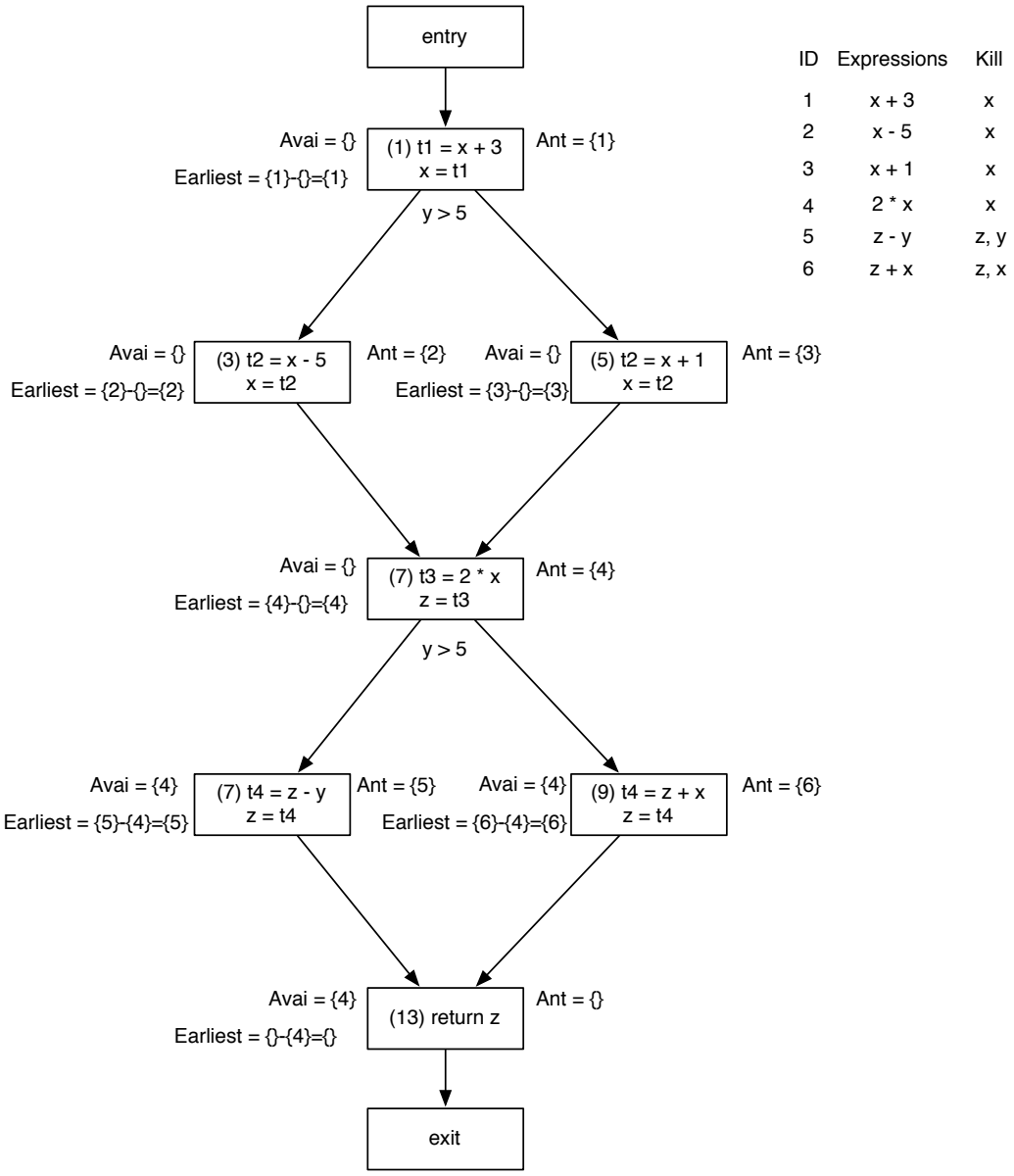
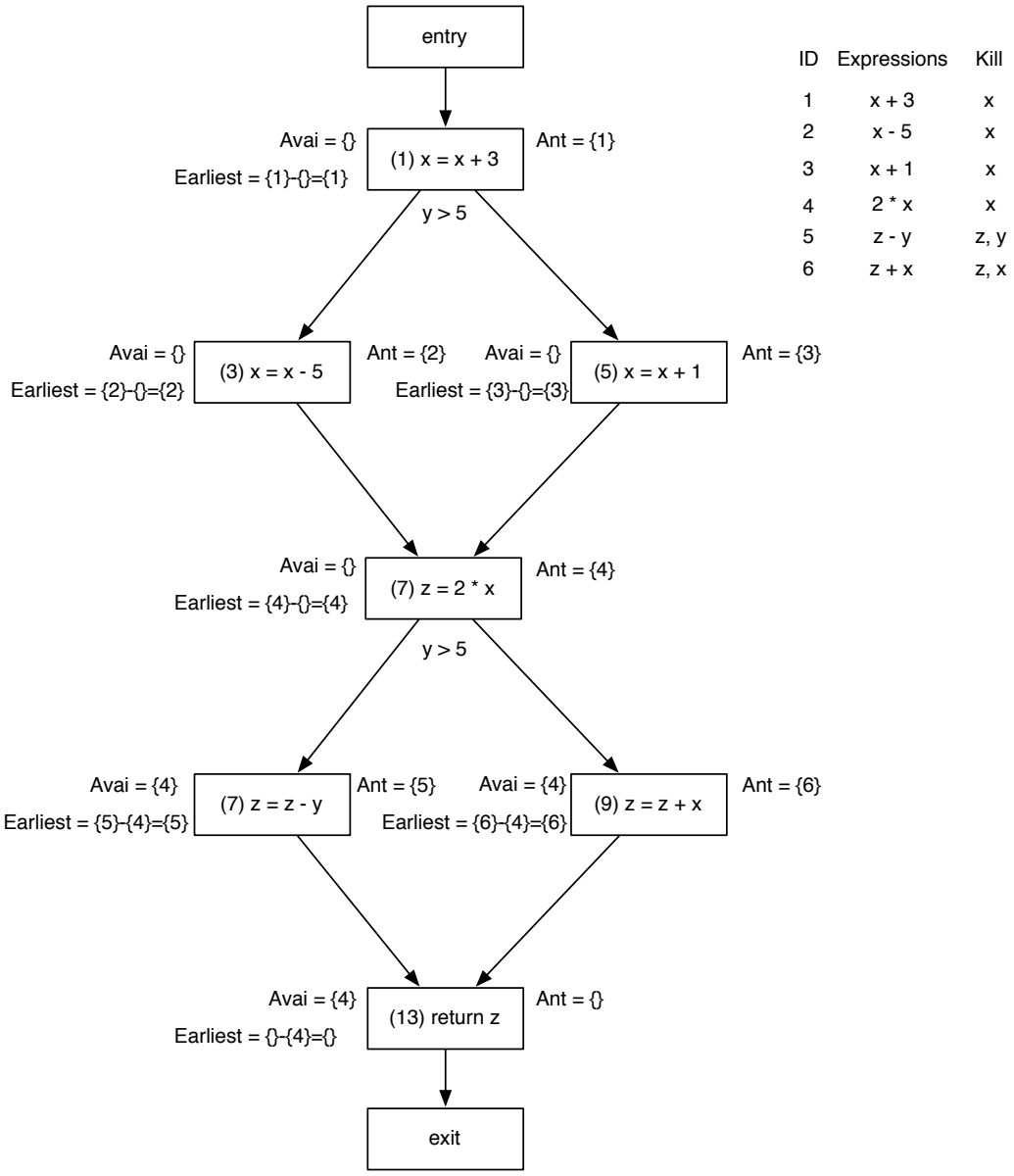


Figure 3: CFG after pass 2: computing `earliest` and adding instructions.



Item 3

Figure 4: CFG after pass 2: constant folding.



1.2 LICM: Loop Invariant Code Motion

Loop invariant instructions:

- S2: `y = 5`
- S3: `q = 7`
- S9: `m = y + 7` (only one reaching definition: S2)
- S12: `r = q + 9` (only one reaching definition: S3)

S10 is not an invariant since g has two reaching definitions (`g = 3` and `g = 4`).

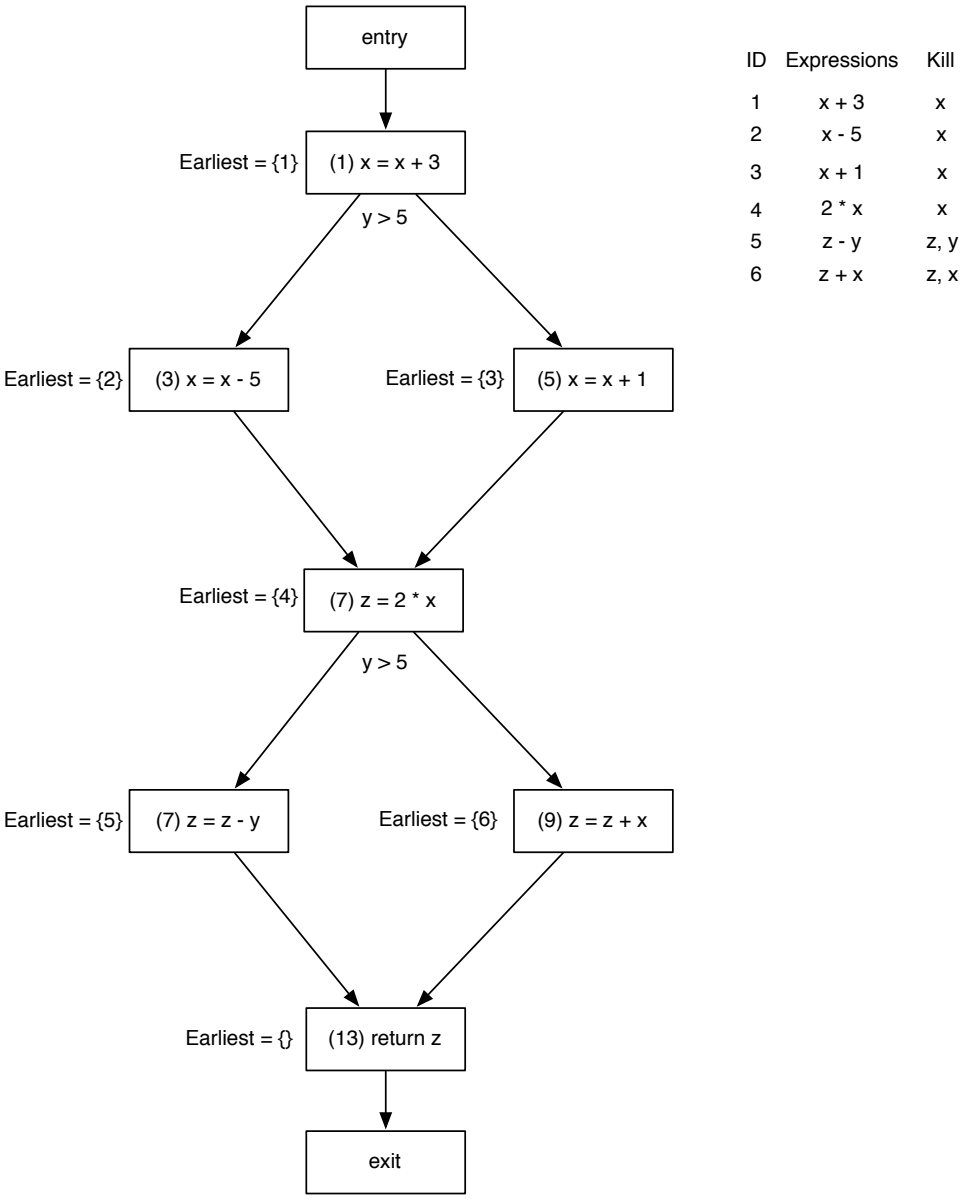
S6 can also be considered a loop invariant, but execution can skip this block and `print` needs to print an undefined x value. So if this instruction was to be moved to the pre-header, it would print 1.

Moved instructions by loop invariant code motion pass:

For each statement s previously listed defining x, we move s to preheader if:

- s is in a block that dominates all exists of the loop,

Figure 5: CFG after pass 2: cleanup.



- x is not defined elsewhere in the loop, and
- s is in a block that dominates all uses of x in the loop.

Since S11 is dead code, we removed this instruction and moved S2 (`y = 5`) to the pre-header.

We cannot move both S9 and S12, since both may not be executed at all, which would jeopardize the program logic when the owner block is not executed.

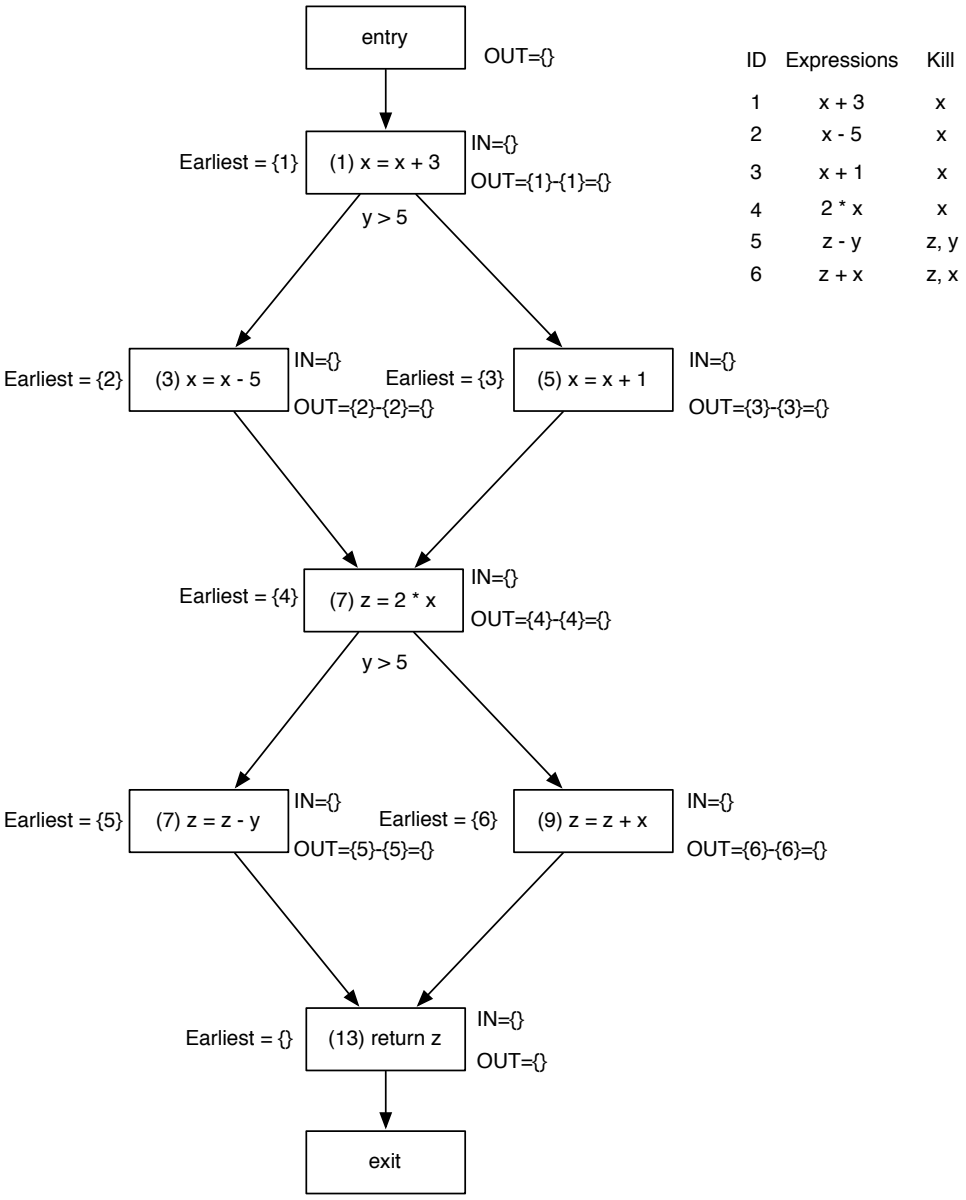
S3 can be moved without problems because the owner block is always be executed and satisfies all the conditions we listed previously.

The final CFG is shown in Fig. 9.

2 Liveness and Reaching Definitions

The implementation of both passes approached the analysis problem using the iterative framework method. The interface to the actual framework was not entirely implemented in a modular fashion, but because of compartmentalization it could easily be modified to be more modular.

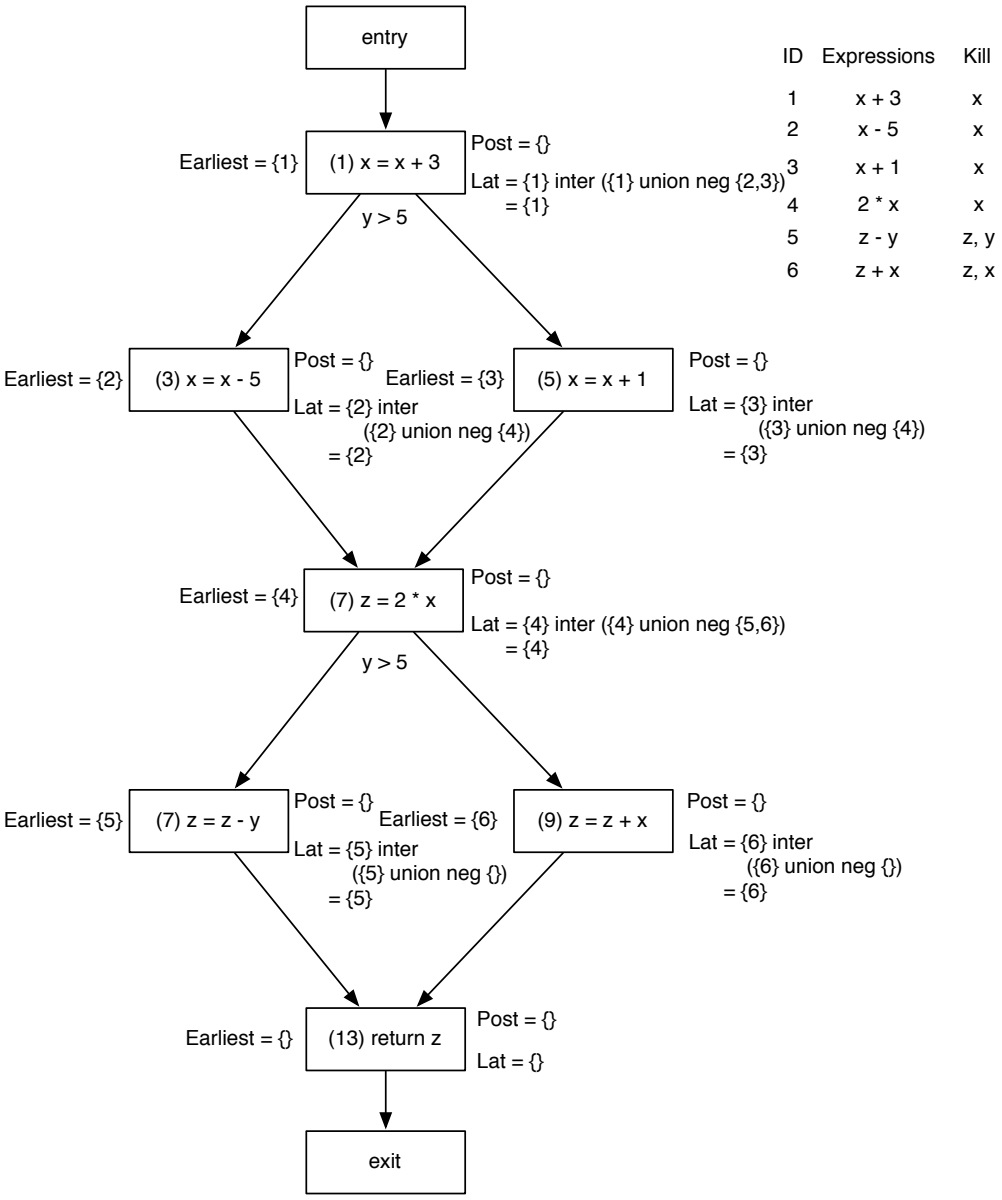
Figure 6: CFG after pass 3: lazy code motion.



One slight bug at the moment is that for reverse traversals program points display a previous result. Unfortunately this could not be resolved in time.

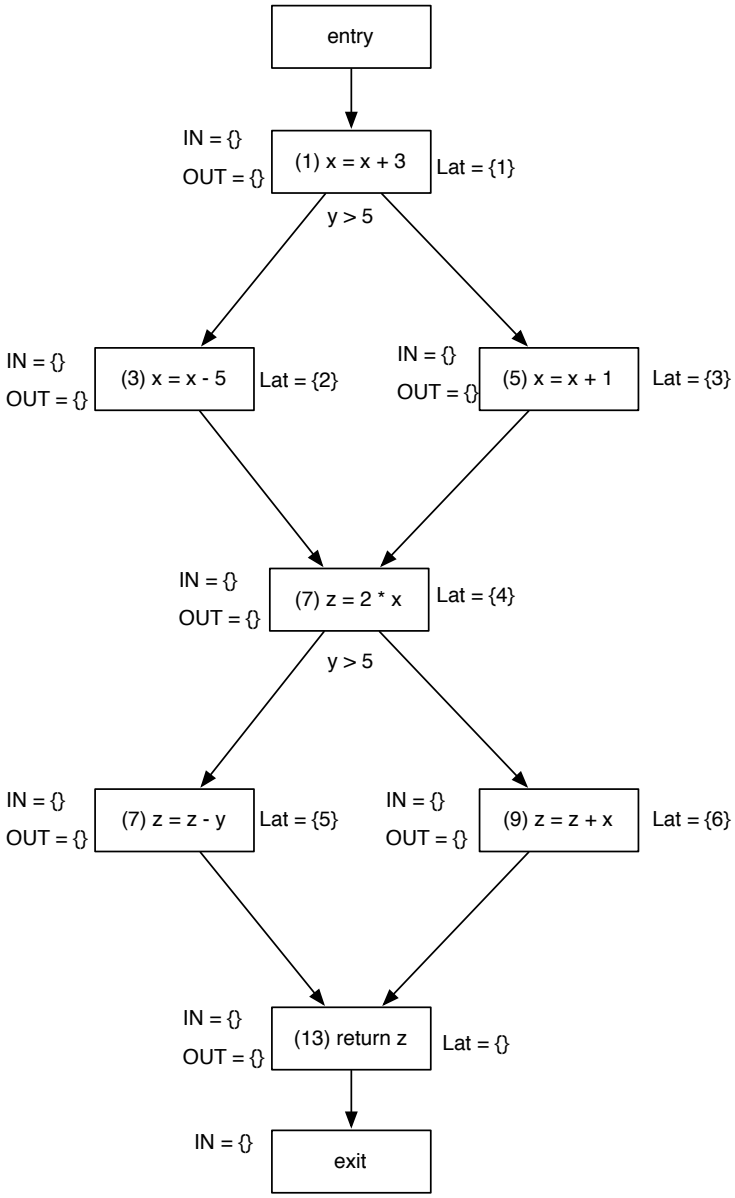
```
#include "llvm/BasicBlock.h"
#include "llvm/Constants.h"
#include "llvm/LLVMContext.h"
#include "llvm/Pass.h"
#include "llvm/User.h"
#include "llvm/Function.h"
#include "llvm/Module.h"
#include "llvm/DerivedTypes.h"
#include "llvm/Instructions.h"
#include "llvm/InstrTypes.h"
#include "llvm/Support/FormattedStream.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/ADT/Twine.h"

#include <ostream>
#include <fstream>
#include <sstream>
#include <iostream>
#include <string>
#include <list>
#include <vector>
#include <map>
```

Figure 7: CFG after pass 3: computing latest.




```
using namespace llvm;
using namespace std;

namespace
{

class Live : public ModulePass
{

    struct cfg {
        list<BasicBlock*> vertices;
        multimap<BasicBlock*, BasicBlock* > bb_edges;
        BasicBlock *start;
        BasicBlock *end;

        // WARNING: This really isn't the right place for this, but we have to
        //          pass the arguments somehow.
        Function *F;

    };

    // At every point and IN/OUT there is a vector corresponding to all values
    // used throughout the program. Depending on the analysis being run the
    // values are flagged accordingly.
    //
    // with this said there also, must be a map between Values->index
```

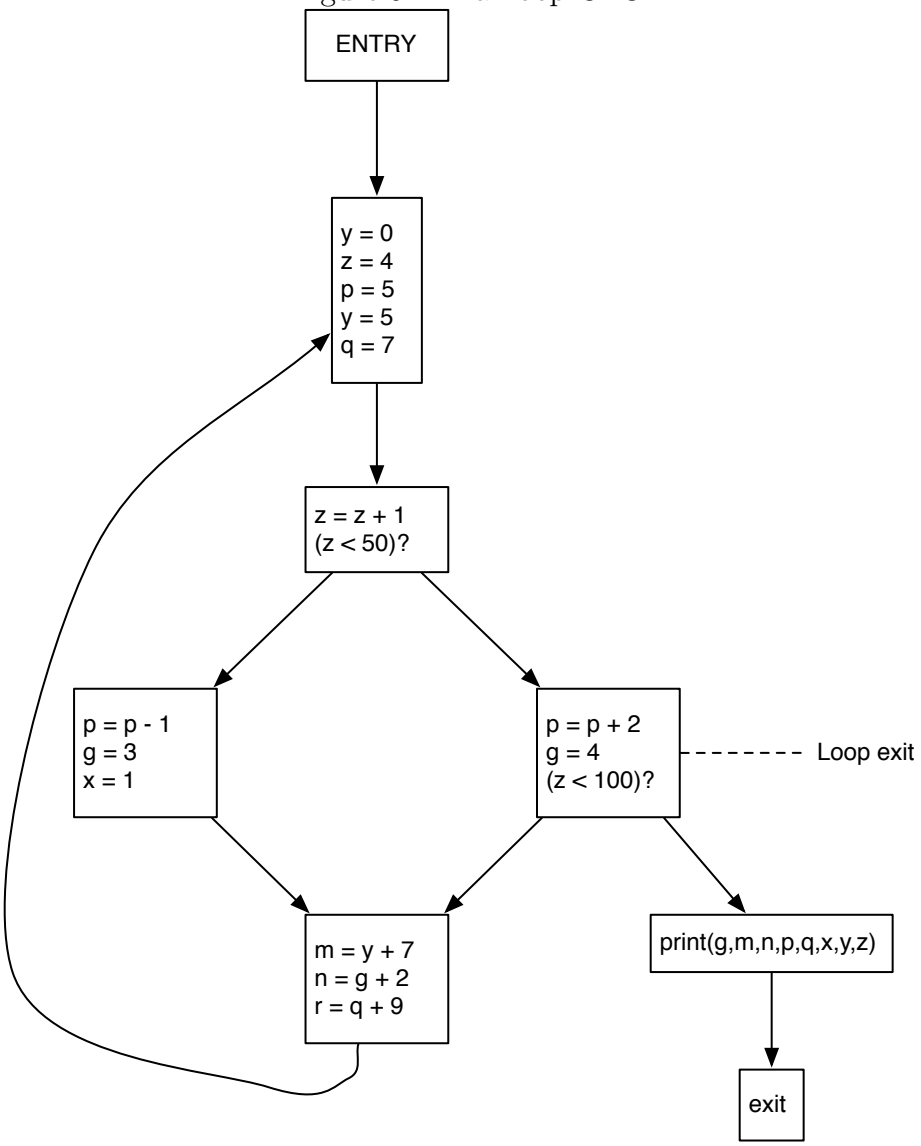
Figure 8: CFG after pass 4: cleaning up.



ID	Expressions	Kill
1	x + 3	x
2	x - 5	x
3	x + 1	x
4	2 * x	x
5	z - y	z, y
6	z + x	z, x

```
    //
    // Additionally, program points are mapped by the instruction after them.
    struct blockPoints{
        vector<bool> *in;
        vector<bool> *out;
        map<Value*, vector<bool>*> programPoints;
    };


public:

        static char ID;

        Live() :
        ModulePass(ID)
        {
        }

        ~Live()
        {
        }

    // We don't modify the program, so we preserve all analyses
    virtual void getAnalysisUsage(AnalysisUsage &AU) const
    {
        AU.setPreservesAll();
```

Figure 9: Final loop CFG.




```
}

virtual bool runOnFunction(Function &F)
{
    cfg *fun_cfg = new cfg;
    map<BlockAddress*,BasicBlock*> block_addr_to_block;

    fun_cfg->start =  BasicBlock::Create(F.getContext(),"start",NULL);
    fun_cfg->end   =  BasicBlock::Create(F.getContext(),"end",NULL);

    // WARNING: This may be on the stack.
    // We want to keep the function in the CFG just so we can grab the arguments.
    fun_cfg->F = &F;

    // Function > GlobalValue > Value
    string name(F.getName().data());

    // First let us build the map of BasicBlocks, and the vertex list.
    for(Function::BasicBlockListType::iterator bl=F.begin(); bl != F.end(); ++bl)
    {
        BlockAddress *ba = BlockAddress::get(bl);
        block_addr_to_block[ba]=bl;
        // WARNING: where is this list being created. The heap, right?
        fun_cfg->vertices.push_back(bl);
    }


    for(Function::BasicBlockListType::iterator bl=F.begin(); bl != F.end(); ++bl)
    {
        BlockAddress *ba = BlockAddress::get(bl);
        TerminatorInst *t = bl-> getTerminator();
```

```
        unsigned num = t->getNumSuccessors();


        if(bl== F.begin())
        {
            pair<BasicBlock*, BasicBlock* > p = pair<BasicBlock*, BasicBlock* >(fun_cfg->start,bl);
            fun_cfg->bb_edges.insert(p);
        }

        if(num == 0)
        {
            pair<BasicBlock*, BasicBlock* > p = pair<BasicBlock*, BasicBlock* >(bl,fun_cfg->end);
            fun_cfg->bb_edges.insert(p);

        }
        else
        {
            for(int i = 0; i < num; i++)
            {
                BasicBlock *term_bb     = t->getSuccessor(i);
                BlockAddress *term_addr = BlockAddress::get(term_bb);

                pair<BasicBlock*, BasicBlock* > p =
                    pair<BasicBlock*, BasicBlock* >(bl,block_addr_to_block[term_addr]);

                fun_cfg->bb_edges.insert(p);

            }
        }

    }

    //printCFG(fun_cfg);

    runIterativeFramework(fun_cfg,false);

    return false;
}

virtual bool printCFG(cfg *scfg)
{
    //print out start and end
    cout<< "Start: " << scfg->start << endl;
    cout<< "End:   " << scfg->end << endl<<endl;

    //print out nodes
    cout << "Vertices:" <<endl;
    for(list<BasicBlock*>::iterator v=scfg->vertices.begin(); v !=scfg->vertices.end();++v)
    {
        cout << *v << endl;
    }



    cout<<endl<<"Edges:" <<endl;
    //print out edges
    for(multimap<BasicBlock*,BasicBlock*>::iterator it=scfg->bb_edges.begin();it!= scfg->bb_edges.end();++it)
        cout << it->first<<" -> "<< it->second <<endl;
    cout <<endl;
    return false;
}

virtual bool printValueNames(map<Value*,unsigned> *valuesToIndex, vector<bool> *v)
{
    vector<bool> *indexCalled = new vector<bool>(v->size());
    cout << "{ ";
    for(map<Value*,unsigned>::iterator it = valuesToIndex->begin(); it != valuesToIndex->end(); ++it)
    {
        if((*v)[it->second])
            if(it->first->hasName())
                cout << " "<<it->first->getName().data();
            else if(!(*indexCalled)[it->second])
            {
                cout << " " << it->second;
                (*indexCalled)[it->second] = true;
            }
    }
    cout << "}"<< endl;
}

// This function prints out the bit code with the program points drawn in.
virtual bool printProgramPoints( Function &F,map<BasicBlock*,blockPoints*> *BBtoBlockPoint,
                                 map<Value*,unsigned> *valuesToIndex)
{
    map<Value*, string> name_map;
    unsigned avail_names=0;
    int point = 0;

    for(Function::BasicBlockListType::iterator bl=F.begin(); bl != F.end(); ++bl)
    {
        cout <<endl <<bl->getName().data() <<":"<<endl;

        cout << "IN: ";
        //printVector((*BBtoBlockPoint)[bl]->in);
        printValueNames(valuesToIndex, (*BBtoBlockPoint)[bl]->in);

        for(BasicBlock::InstListType::iterator inst=bl->begin(); inst != bl->end(); ++inst)
        {
```

```
            //string s = "";
            //raw_string_ostream *ss = new raw_string_ostream(s);
            //inst->print(*ss);

            // These operations return values.
            // 0. store: Unaryinst
            // 1. binary ops
            // 2. cast ops
            // 3. PHI Nodes have values, but we don't want points infront of them.
            if(isa<PHINode>(inst)|| isa<BinaryOperator>(inst) ||
               isa<UnaryInstruction>(inst)||isa<CmpInst>(inst))
            {
                // we need to give the assignment a name:
                std::stringstream out;
                out << ++avail_names;
                name_map[inst]=string(out.str());

                // we want to flag the phi node so we don't have a point.
                if(!isa<PHINode>(inst))
                {
                    //cout << "p"<<point++ <<endl;
                    printValueNames(valuesToIndex, (*BBtoBlockPoint)[bl]->programPoints[inst]);
                    cout << endl;
                }
                cout << "\t"<<name_map[inst]<<" = "<< inst->getOpcodeName();
            }
            // These consume but do not produce.
            // Well actually the call could if they were pointers.
            else if(isa<TerminatorInst>(inst)||isa<StoreInst>(inst)||isa<CallInst>(inst))
            {
                //cout << "p"  << point++ <<endl;
                printValueNames(valuesToIndex, (*BBtoBlockPoint)[bl]->programPoints[inst]);
                cout << "\t" << inst->getOpcodeName();
            }
            // we just spit out everything else.
            else
            {
                cout << "\tINVALID OP"<< inst->getOpcodeName();
            }

            for(User::op_iterator oi= inst->op_begin(), oe= inst->op_end(); oi!=oe ;++oi)
            {

                Value *v = oi->get();

                // Do we need these values?
                if(isa<Instruction>(v)||isa<Argument>(v))
                {
                    // because assignments are not given names until code
                    // generation. we will just give them some temp values.
                    if(name_map.find(v) == name_map.end())
                        if(v->hasName())
                            name_map[v]=v->getName().data();
                        else
                        {
                            std::stringstream out;
                            out << ++avail_names;
                            name_map[v]=string(out.str());
                        }


                    cout<< " " <<name_map[v];
                }
            }

            cout<< endl;
            //inst->print(*ss);

            if(isa<TerminatorInst>(inst))
            {
                cout << "OUT: ";
                //printVector((*BBtoBlockPoint)[bl]->out);
                printValueNames(valuesToIndex, (*BBtoBlockPoint)[bl]->out);
            }

        }
    }
}

// This function takes in a pointer to a CFG and a pointer to an empty block map and
// contstructs the mapping of basicblocks to blockpoints.
//
// Additionally, it initializes all of the vectors. So, it must keep track of
// the number of values.
//
// Lastly, it produces the mapping between values and their vector index.
virtual bool buildBlockPointMap( cfg *CFG, map<BasicBlock*,blockPoints*> *BBtoBlockPoint,
                                 map<Value*,unsigned> *valuesToIndex )
{
    //map<Value*, string> name_map;
    unsigned index=0;
    int point = 0;


    // TODO: I need to iterate through the Function and make the arguments blockPoints for Start.
    blockPoints *start_bp = new blockPoints;
    blockPoints *end_bp = new blockPoints;
```

```
    (*BBtoBlockPoint)[CFG->start] = start_bp;
    (*BBtoBlockPoint)[CFG->end] = end_bp;

    for(list<BasicBlock*>::iterator block=CFG->vertices.begin(); block != CFG->vertices.end(); ++block)
    {
        BasicBlock *bl  = *block;
        blockPoints *bp = new blockPoints;

        (*BBtoBlockPoint)[bl]=bp;


        for(BasicBlock::InstListType::iterator inst=bl->begin(); inst != bl->end(); ++inst)
        {
            // These operations return values.
            // 0. store: Unaryinst
            // 1. binary ops
            // 2. cast ops
            // 3. PHI Nodes have values, but we don't want points infront of them.
            if(isa<PHINode>(inst)|| isa<BinaryOperator>(inst) ||
               isa<UnaryInstruction>(inst)||isa<CmpInst>(inst))
            {
                //names will become indices.
                (*valuesToIndex)[inst]=index++;

                // we want to flag the phi node so we don't have a point.
                if(!isa<PHINode>(inst))
                    // we push the point on a list so we can initialize them later.
                    bp->programPoints[inst] = NULL;

            }
            // These consume but do not produce.
            // Well actually the call could if they were pointers.
            else if(isa<TerminatorInst>(inst)||isa<StoreInst>(inst)||isa<CallInst>(inst))
            {
                bp->programPoints[inst] = NULL;
            }
            // we just spit out everything else.
            else
            {
                cout << "\tHRM"<< inst->getOpcodeName();
            }

            for(User::op_iterator oi= inst->op_begin(), oe= inst->op_end(); oi!=oe ;++oi)
            {

                Value *v = oi->get();

                // Do we need these values?
                if(isa<Instruction>(v)||isa<Argument>(v))
                {

                    // if the argument is not in the value map then let's add it.
                    if(valuesToIndex->find(v) == valuesToIndex->end())
                        (*valuesToIndex)[v]=index++;
                }
            }

        }

    }


    // We are going to give the arguments an index into the vector.
    for(Function::ArgumentListType::iterator ag=CFG->F->arg_begin(); ag != CFG->F->arg_end(); ++ag)
        if(valuesToIndex->find(ag) == valuesToIndex->end())
            (*valuesToIndex)[ag]=index++;

    // Note: Need to initialize start and end which means I need to grab
    //       the function arguments. This changes everything because it
    //       adds more positions to the vectors.
    start_bp->in  = NULL;
    start_bp->out = new vector<bool>(valuesToIndex->size());

    end_bp->in  = new vector<bool>(valuesToIndex->size());
    end_bp->out = NULL;

    // we know all of the values, now we can initializes all the vectors on the heap.
    for(list<BasicBlock*>::iterator block=CFG->vertices.begin(); block != CFG->vertices.end(); ++block)
    {
        BasicBlock *bl  = *block;
        blockPoints *bp = (*BBtoBlockPoint)[bl];


        bp->in = new vector<bool>(valuesToIndex->size());
        bp->out = new vector<bool>(valuesToIndex->size());

        for(map<Value*, vector<bool>*>::iterator it = bp->programPoints.begin(); it != bp->programPoints.end(); ++it)
        {
            bp->programPoints[it->first] = new vector<bool>(valuesToIndex->size());
        }

        //cout << "pPS: "<< bp->programPoints.size() <<endl;
    }

}
```

```
// TODO: I need more arguments here
// For now let's just implement reaching defs to make sure it works.
virtual bool runIterativeFramework(cfg *CFG, bool fromTop /*, transfer_function, meet_op, set_boundary, set_initial*/)
{
    // need a modified blocks list for the while loop.
    bool somethingModified=true;

    map<BasicBlock*,blockPoints*> BBtoBlockPoint;
    map<Value*,unsigned> valuesToIndex;

    buildBlockPointMap(CFG,&BBtoBlockPoint ,&valuesToIndex);



    // Initialize Boundary conditions.
    // +we want to set out[entry]

    if(fromTop)
    {
        setEmpty(BBtoBlockPoint[CFG->start]->out); //reach-def
        // but actually we do want the args
        for(Function::ArgumentListType::iterator ag=CFG->F->arg_begin(); ag != CFG->F->arg_end(); ++ag)
            (*BBtoBlockPoint[CFG->start]->out)[valuesToIndex[ag]] = true;

    }
    else
    {
    }

    //if(fromTop)
    //    modifiedList.push_back(CFG->start);


    // Init the outs/ins depending on direction.
    for(list<BasicBlock*>::iterator block=CFG->vertices.begin(); block != CFG->vertices.end(); ++block)
        setEmpty(BBtoBlockPoint[*block]->in); //live

    // Iterate
    while(somethingModified)
    {
        somethingModified=false;

        for(list<BasicBlock*>::iterator block=CFG->vertices.begin(); block != CFG->vertices.end(); ++block)
        {
            // for each BB other than entry, so including exit?
            if(fromTop)
            {
                vector<bool> tmp = *((BBtoBlockPoint[*block]->out));
                // We want to union all of the outs of the preds
                // in[B] = U (out[p])
                for ( multimap<BasicBlock*, BasicBlock* >::iterator it=CFG->bb_edges.begin() ; it != CFG->bb_edges.end(); it++
                    if(it->second == *block)
                    {
                        //cout << "predL " << it->first << ": ";
                        //printVector(BBtoBlockPoint[it->first]->out);

                        // in[b] = in[b] U (out[p])
                        unionVect((BBtoBlockPoint[*block]->in),(BBtoBlockPoint[it->first]->out));
                    }

                // out = F_b(in[B])
                defTransfer(BBtoBlockPoint[*block], &valuesToIndex);

                // if tmp differs from  out then we flag a modify
                if(tmp!=*((BBtoBlockPoint[*block]->out)))
                    somethingModified = true;

                // printVector(BBtoBlockPoint[*block]->in);

            }
            else // not from top
            {

                vector<bool> tmp = *((BBtoBlockPoint[*block]->in));//
                // We want to union all of the outs of the preds
                // in[B] = U (out[p])
                for ( multimap<BasicBlock*, BasicBlock* >::iterator it=CFG->bb_edges.begin() ; it != CFG->bb_edges.end(); it++
                    if(it->first == *block) //
                    {
                        unionVect((BBtoBlockPoint[*block]->out),(BBtoBlockPoint[it->first]->in));//
                    }

                // out = F_b(in[B])
                liveTransfer(BBtoBlockPoint[*block], &valuesToIndex);

                // if tmp differs from  out then we flag a modify
                if(tmp!=*((BBtoBlockPoint[*block]->in)))//
                    somethingModified = true;


            }
        }

    }
```

```
        printProgramPoints(*(CFG->F),&BBtoBlockPoint ,&valuesToIndex);

    }

    virtual bool liveTransfer(blockPoints *bp, map<Value*,unsigned> *valuesToIndex)
    {
        vector<bool> *v_old = bp->out;

        for(map<Value*, vector<bool>*>::reverse_iterator it = bp->programPoints.rbegin();
            it != bp->programPoints.rend(); ++it)
        {
            vector<bool> *v_temp = new vector<bool>(it->second->size());
            vector<bool> *v_temp2 = new vector<bool>(it->second->size());
            vector<bool> *v_kill = new vector<bool>(it->second->size());

            // kill
            if(isa<Instruction>(it->first)||isa<Argument>(it->first))
            {
//                cout << "KILL: "<<(*valuesToIndex)[it->first] <<endl;
                (*v_kill)[(*valuesToIndex)[it->first]] = true;
            }

            // gen/use
            Instruction *inst = (Instruction *)it->first;
            for(User::op_iterator oi= inst->op_begin(), oe= inst->op_end(); oi!=oe ;++oi)
                if(isa<Instruction>(oi)||isa<Argument>(oi))
                    (*v_temp)[(*valuesToIndex)[*oi]] = true;

            // it = gen U (v_old - kill)
            removeElements(v_temp2,v_old,v_kill);

            unionVect(v_temp,v_temp2);
            (*it->second) = *v_temp;

            v_old = (it->second);

            //cout << "ppp "<< it->first;
            //printVector(it->second);

            delete v_temp;
            delete v_temp2;
            delete v_kill;



        }
        (*bp->in)= *v_old;
    }

    // At the block
    // TODO: still need to think about Phi handling
    virtual bool defTransfer(blockPoints *bp, map<Value*,unsigned> *valuesToIndex)
    {
        vector<bool> *v_old = bp->in;

        for(map<Value*, vector<bool>*>::iterator it = bp->programPoints.begin();
            it != bp->programPoints.end(); ++it)
        {
            vector<bool> *v_temp = new vector<bool>(it->second->size());
            vector<bool> *v_temp2 = new vector<bool>(it->second->size());
            vector<bool> *v_kill = new vector<bool>(it->second->size());

            //gen
            if(isa<Instruction>(it->first)||isa<Argument>(it->first))
                (*v_temp)[(*valuesToIndex)[it->first]] = true;

            //kill: will only kill values at PHI
            if(isa<PHINode>(it->first))
            {
                PHINode *inst = (PHINode *)it->first;
                for(User::op_iterator oi= inst->op_begin(), oe= inst->op_end(); oi!=oe ;++oi)
                {
                    (*v_kill)[(*valuesToIndex)[inst]] = true;
                }
            }
            // it = gen U (v_old - kill)
            removeElements(v_temp2,v_old,v_kill);

            unionVect(v_temp,v_temp2);
            (*it->second) = *v_temp;

            v_old = (it->second);

            //cout << "ppp "<< it->first;
            //printVector(it->second);

            delete v_temp;
            delete v_temp2;
            delete v_kill;
        }

        // set out to the last point
        (*bp->out)= *v_old;

    }


    virtual bool removeElements(vector<bool> *vr,vector<bool> *v1,vector<bool> *v2)
```

```
    {
        for(int i=0; i < v1->size();i++)
            if((*v2)[i])
                (*vr)[i]=false;
            else
                (*vr)[i] = (*v1)[i];
    }

    virtual bool setEmpty(vector<bool> *v)
    {
        for(int i=0; i < v->size();i++)
            (*v)[i]=false;
    }

    // v1 <- v1 or v2
    virtual bool unionVect(vector<bool> *v1,vector<bool> *v2)
    {
        for(int i=0; i < v1->size();i++)
            (*v1)[i]=(*v1)[i]|(*v2)[i];
    }


    virtual bool printVector(vector<bool> *v)
    {
        cout << "[";
        for(int i=0; i < v->size();i++)
            cout << " " <<(*v)[i];
        cout << "]" << endl;

    }

    // Go through each line and if value matches a point print out the values
    virtual bool printReachDefResults()
    {
    }

    virtual bool runOnModule(Module& M)
    {


        for (Module::iterator MI = M.begin(), ME = M.end(); MI != ME; ++MI)
            runOnFunction(*MI);



        return false;
    }
};

char Live::ID = 0;
RegisterPass<Live> X("live", "15745: Iterative live variable Analysis");

}

#include "llvm/BasicBlock.h"
#include "llvm/Constants.h"
#include "llvm/LLVMContext.h"
#include "llvm/Pass.h"
#include "llvm/User.h"
#include "llvm/Function.h"
#include "llvm/Module.h"
#include "llvm/DerivedTypes.h"
#include "llvm/Instructions.h"
#include "llvm/InstrTypes.h"
#include "llvm/Support/FormattedStream.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/ADT/Twine.h"

#include <ostream>
#include <fstream>
#include <sstream>
#include <iostream>
#include <string>
#include <list>
#include <vector>
#include <map>

using namespace llvm;
using namespace std;

namespace
{

class ReachingDef : public ModulePass
{

    struct cfg {
        list<BasicBlock*> vertices;
        multimap<BasicBlock*, BasicBlock* > bb_edges;
        BasicBlock *start;
        BasicBlock *end;

        // WARNING: This really isn't the right place for this, but we have to
        //          pass the arguments somehow.
        Function *F;

    };

    // At every point and IN/OUT there is a vector corresponding to all values
    // used throughout the program. Depending on the analysis being run the
```

```
    // values are flagged accordingly.
    //
    // with this said there also, must be a map between Values->index
    //
    // Additionally, program points are mapped by the instruction after them.
    struct blockPoints{
        vector<bool> *in;
        vector<bool> *out;
        map<Value*, vector<bool>*> programPoints;
    };


public:

        static char ID;

        ReachingDef() :
        ModulePass(ID)
        {
        }

        ~ReachingDef()
        {
        }

    // We don't modify the program, so we preserve all analyses
    virtual void getAnalysisUsage(AnalysisUsage &AU) const
    {
        AU.setPreservesAll();
    }

    virtual bool runOnFunction(Function &F)
    {
        cfg *fun_cfg = new cfg;
        map<BlockAddress*,BasicBlock*> block_addr_to_block;

        fun_cfg->start =  BasicBlock::Create(F.getContext(),"start",NULL);
        fun_cfg->end   =  BasicBlock::Create(F.getContext(),"end",NULL);

        // WARNING: This may be on the stack.
        // We want to keep the function in the CFG just so we can grab the arguments.
        fun_cfg->F = &F;

        // Function > GlobalValue > Value
        string name(F.getName().data());

        // First let us build the map of BasicBlocks, and the vertex list.
        for(Function::BasicBlockListType::iterator bl=F.begin(); bl != F.end(); ++bl)
        {
            BlockAddress *ba = BlockAddress::get(bl);
            block_addr_to_block[ba]=bl;
            // WARNING: where is this list being created. The heap, right?
            fun_cfg->vertices.push_back(bl);
        }


        for(Function::BasicBlockListType::iterator bl=F.begin(); bl != F.end(); ++bl)
        {
            BlockAddress *ba = BlockAddress::get(bl);
            TerminatorInst *t = bl-> getTerminator();
            unsigned num = t->getNumSuccessors();


            if(bl== F.begin())
            {
                pair<BasicBlock*, BasicBlock* > p = pair<BasicBlock*, BasicBlock* >(fun_cfg->start,bl);
                fun_cfg->bb_edges.insert(p);
            }

            if(num == 0)
            {
                pair<BasicBlock*, BasicBlock* > p = pair<BasicBlock*, BasicBlock* >(bl,fun_cfg->end);
                fun_cfg->bb_edges.insert(p);

            }
            else
            {
                for(int i = 0; i < num; i++)
                {
                    BasicBlock *term_bb     = t->getSuccessor(i);
                    BlockAddress *term_addr = BlockAddress::get(term_bb);

                    pair<BasicBlock*, BasicBlock* > p =
                        pair<BasicBlock*, BasicBlock* >(bl,block_addr_to_block[term_addr]);

                    fun_cfg->bb_edges.insert(p);

                }
            }

        }

        //printCFG(fun_cfg);

        runIterativeFramework(fun_cfg,true);

        return false;
    }
```

```
virtual bool printCFG(cfg *scfg)
{
    //print out start and end
    cout<< "Start: " << scfg->start << endl;
    cout<< "End:   " << scfg->end << endl<<endl;

    //print out nodes
    cout << "Vertices:" <<endl;
    for(list<BasicBlock*>::iterator v=scfg->vertices.begin(); v !=scfg->vertices.end();++v)
    {
        cout << *v << endl;
    }


    cout<<endl<<"Edges:" <<endl;
    //print out edges
    for(multimap<BasicBlock*,BasicBlock*>::iterator it=scfg->bb_edges.begin();it!= scfg->bb_edges.end();++it)
        cout << it->first <<" -> "<< it->second <<endl;
    cout <<endl;
    return false;
}

virtual bool printValueNames(map<Value*,unsigned> *valuesToIndex, vector<bool> *v)
{
    vector<bool> *indexCalled = new vector<bool>(v->size());
    cout << "{ ";
    for(map<Value*,unsigned>::iterator it = valuesToIndex->begin(); it != valuesToIndex->end(); ++it)
    {
        if((*v)[it->second])
            if(it->first->hasName())
                cout << " "<<it->first->getName().data();
            else if(!(*indexCalled)[it->second])
            {
                cout << " " << it->second;
                (*indexCalled)[it->second] = true;
            }
    }
    cout << "}"<< endl;
}

// This function prints out the bit code with the program points drawn in.
virtual bool printProgramPoints( Function &F,map<BasicBlock*,blockPoints*> *BBtoBlockPoint,
                                 map<Value*,unsigned> *valuesToIndex)
{
    map<Value*, string> name_map;
    unsigned avail_names=0;
    int point = 0;

    for(Function::BasicBlockListType::iterator bl=F.begin(); bl != F.end(); ++bl)
    {
        cout <<endl <<bl->getName().data() <<":"<<endl;

        cout << "IN: ";
        //printVector((*BBtoBlockPoint)[bl]->in);
        printValueNames(valuesToIndex, (*BBtoBlockPoint)[bl]->in);

        for(BasicBlock::InstListType::iterator inst=bl->begin(); inst != bl->end(); ++inst)
        {

            //string s = "";
            //raw_string_ostream *ss = new raw_string_ostream(s);
            //inst->print(*ss);

            // These operations return values.
            // 0. store: Unaryinst
            // 1. binary ops
            // 2. cast ops
            // 3. PHI Nodes have values, but we don't want points infront of them.
            if(isa<PHINode>(inst)|| isa<BinaryOperator>(inst) ||
               isa<UnaryInstruction>(inst)||isa<CmpInst>(inst))
            {
                // we need to give the assignment a name:
                std::stringstream out;
                out << ++avail_names;
                name_map[inst]=string(out.str());

                // we want to flag the phi node so we don't have a point.
                if(!isa<PHINode>(inst))
                {
                    //cout << "p"<<point++ <<endl;
                    printValueNames(valuesToIndex, (*BBtoBlockPoint)[bl]->programPoints[inst]);
                    cout << endl;
                }
                cout << "\t"<<name_map[inst]<<" = "<< inst->getOpcodeName();
            }
            // These consume but do not produce.
            // Well actually the call could if they were pointers.
            else if(isa<TerminatorInst>(inst)||isa<StoreInst>(inst)||isa<CallInst>(inst))
            {
                //cout << "p"  << point++ <<endl;
                printValueNames(valuesToIndex, (*BBtoBlockPoint)[bl]->programPoints[inst]);
                cout << "\t" << inst->getOpcodeName();
            }
            // we just spit out everything else.
            else
            {
                cout << "\tINVALID OP"<< inst->getOpcodeName();
```

```
            }

            for(User::op_iterator oi= inst->op_begin(), oe= inst->op_end(); oi!=oe ;++oi)
            {

                Value *v = oi->get();

                // Do we need these values?
                if(isa<Instruction>(v)||isa<Argument>(v))
                {
                    // because assignments are not given names until code
                    // generation. we will just give them some temp values.
                    if(name_map.find(v) == name_map.end())
                        if(v->hasName())
                            name_map[v]=v->getName().data();
                        else
                        {
                            std::stringstream out;
                            out << ++avail_names;
                            name_map[v]=string(out.str());
                        }


                    cout<< " " <<name_map[v];
                }
            }

            cout<< endl;
            //inst->print(*ss);

            if(isa<TerminatorInst>(inst))
            {
                cout << "OUT: ";
                //printVector((*BBtoBlockPoint)[bl]->out);
                printValueNames(valuesToIndex, (*BBtoBlockPoint)[bl]->out);
            }

        }
    }
}

// This function takes in a pointer to a CFG and a pointer to an empty block map and
// contstructs the mapping of basicblocks to blockpoints.
//
// Additionally, it initializes all of the vectors. So, it must keep track of
// the number of values.
//
// Lastly, it produces the mapping between values and their vector index.
virtual bool buildBlockPointMap( cfg *CFG, map<BasicBlock*,blockPoints*> *BBtoBlockPoint,
                                 map<Value*,unsigned> *valuesToIndex )
{
    //map<Value*, string> name_map;
    unsigned index=0;
    int point = 0;


    // TODO: I need to iterate through the Function and make the arguments blockPoints for Start.
    blockPoints *start_bp = new blockPoints;
    blockPoints *end_bp = new blockPoints;

    (*BBtoBlockPoint)[CFG->start] = start_bp;
    (*BBtoBlockPoint)[CFG->end] = end_bp;

    for(list<BasicBlock*>::iterator block=CFG->vertices.begin(); block != CFG->vertices.end(); ++block)
    {
        BasicBlock *bl  = *block;
        blockPoints *bp = new blockPoints;

        (*BBtoBlockPoint)[bl]=bp;


        for(BasicBlock::InstListType::iterator inst=bl->begin(); inst != bl->end(); ++inst)
        {
            // These operations return values.
            // 0. store: Unaryinst
            // 1. binary ops
            // 2. cast ops
            // 3. PHI Nodes have values, but we don't want points infront of them.
            if(isa<PHINode>(inst)|| isa<BinaryOperator>(inst) ||
               isa<UnaryInstruction>(inst)||isa<CmpInst>(inst))
            {
                //names will become indices.
                (*valuesToIndex)[inst]=index++;

                // we want to flag the phi node so we don't have a point.
                if(!isa<PHINode>(inst))
                    // we push the point on a list so we can initialize them later.
                    bp->programPoints[inst] = NULL;

            }
            // These consume but do not produce.
            // Well actually the call could if they were pointers.
            else if(isa<TerminatorInst>(inst)||isa<StoreInst>(inst)||isa<CallInst>(inst))
            {
                bp->programPoints[inst] = NULL;
            }
            // we just spit out everything else.
            else
```

```
            {
                cout << "\tHRM"<< inst->getOpcodeName();
            }

            for(User::op_iterator oi= inst->op_begin(), oe= inst->op_end(); oi!=oe ;++oi)
            {

                Value *v = oi->get();

                // Do we need these values?
                if(isa<Instruction>(v)||isa<Argument>(v))
                {

                    // if the argument is not in the value map then let's add it.
                    if(valuesToIndex->find(v) == valuesToIndex->end())
                        (*valuesToIndex)[v]=index++;
                }
            }

        }

    }


    // We are going to give the arguments an index into the vector.
    for(Function::ArgumentListType::iterator ag=CFG->F->arg_begin(); ag != CFG->F->arg_end(); ++ag)
        if(valuesToIndex->find(ag) == valuesToIndex->end())
            (*valuesToIndex)[ag]=index++;

    // Note: Need to initialize start and end which means I need to grab
    //       the function arguments. This changes everything because it
    //       adds more positions to the vectors.
    start_bp->in  = NULL;
    start_bp->out = new vector<bool>(valuesToIndex->size());

    end_bp->in  = new vector<bool>(valuesToIndex->size());
    end_bp->out = NULL;

    // we know all of the values, now we can initializes all the vectors on the heap.
    for(list<BasicBlock*>::iterator block=CFG->vertices.begin(); block != CFG->vertices.end(); ++block)
    {
        BasicBlock *bl  = *block;
        blockPoints *bp = (*BBtoBlockPoint)[bl];


        bp->in = new vector<bool>(valuesToIndex->size());
        bp->out = new vector<bool>(valuesToIndex->size());

        for(map<Value*, vector<bool>*>::iterator it = bp->programPoints.begin(); it != bp->programPoints.end(); ++it)
        {
            bp->programPoints[it->first] = new vector<bool>(valuesToIndex->size());
        }

        //cout << "pPS: "<< bp->programPoints.size() <<endl;
    }


}


// TODO: I need more arguments here
// For now let's just implement reaching defs to make sure it works.
virtual bool runIterativeFramework(cfg *CFG, bool fromTop /*, transfer_function, meet_op, set_boundary, set_initial*/)
{
    // need a modified blocks list for the while loop.
    bool somethingModified=true;

    map<BasicBlock*,blockPoints*> BBtoBlockPoint;
    map<Value*,unsigned> valuesToIndex;

    buildBlockPointMap(CFG,&BBtoBlockPoint ,&valuesToIndex);



    // Initialize Boundary conditions.
    // +we want to set out[entry]

    if(fromTop)
    {
        setEmpty(BBtoBlockPoint[CFG->start]->out); //reach-def
        // but actually we do want the args
        for(Function::ArgumentListType::iterator ag=CFG->F->arg_begin(); ag != CFG->F->arg_end(); ++ag)
            (*BBtoBlockPoint[CFG->start]->out)[valuesToIndex[ag]] = true;

    }
    else
    {
    }

    //if(fromTop)
    //    modifiedList.push_back(CFG->start);



    // Init the outs/ins depending on direction.
    for(list<BasicBlock*>::iterator block=CFG->vertices.begin(); block != CFG->vertices.end(); ++block)
        setEmpty(BBtoBlockPoint[*block]->out); //reach-def
```

```
    // Iterate
    while(somethingModified)
    {
        somethingModified=false;

        for(list<BasicBlock*>::iterator block=CFG->vertices.begin(); block != CFG->vertices.end(); ++block)
        {
            // for each BB other than entry, so including exit?
            if(fromTop)
            {
                vector<bool> tmp = *((BBtoBlockPoint[*block]->out));
                // We want to union all of the outs of the preds
                // in[B] = U (out[p])
                for ( multimap<BasicBlock*, BasicBlock* >::iterator it=CFG->bb_edges.begin() ; it != CFG->bb_edges.end(); it++
                    if(it->second == *block)
                    {
                        //cout << "predL " << it->first << ": ";
                        //printVector(BBtoBlockPoint[it->first]->out);

                        // in[b] = in[b] U (out[p])
                        unionVect((BBtoBlockPoint[*block]->in),(BBtoBlockPoint[it->first]->out));
                    }

                //tmp = out[b] we want to detect changes

                // out = F_b(in[B])
                defTransfer(BBtoBlockPoint[*block], &valuesToIndex);

                // if tmp differs from  out then we flag a modify
                if(tmp!=*((BBtoBlockPoint[*block]->out)))
                    somethingModified = true;

                // printVector(BBtoBlockPoint[*block]->in);

            }
            else // not from top
            {
            }
        }

    }

    printProgramPoints(*(CFG->F),&BBtoBlockPoint ,&valuesToIndex);

}

// At the block
// TODO: still need to think about Phi handling
virtual bool defTransfer(blockPoints *bp, map<Value*,unsigned> *valuesToIndex)
{
    vector<bool> *v_old = bp->in;

    for(map<Value*, vector<bool>*>::iterator it = bp->programPoints.begin();
        it != bp->programPoints.end(); ++it)
    {
        vector<bool> *v_temp = new vector<bool>(it->second->size());
        vector<bool> *v_temp2 = new vector<bool>(it->second->size());
        vector<bool> *v_kill = new vector<bool>(it->second->size());

        //gen
        if(isa<Instruction>(it->first)||isa<Argument>(it->first))
            (*v_temp)[(*valuesToIndex)[it->first]] = true;

        //kill: will only kill values at PHI
        if(isa<PHINode>(it->first))
        {
            PHINode *inst = (PHINode *)it->first;
            for(User::op_iterator oi= inst->op_begin(), oe= inst->op_end(); oi!=oe ;++oi)
            {
                (*v_kill)[(*valuesToIndex)[inst]] = true;
            }
        }
        // it = gen U (v_old - kill)
        removeElements(v_temp2,v_old,v_kill);

        unionVect(v_temp,v_temp2);
        (*it->second) = *v_temp;

        v_old = (it->second);

        //cout << "ppp "<< it->first;
        //printVector(it->second);

        delete v_temp;
        delete v_temp2;
        delete v_kill;
    }

    // set out to the last point
    (*bp->out)= *v_old;

}


virtual bool removeElements(vector<bool> *vr,vector<bool> *v1,vector<bool> *v2)
{
    for(int i=0; i < v1->size();i++)
        if((*v2)[i])
```

```
                (*vr)[i]=false;
            else
                (*vr)[i] = (*v1)[i];
    }

    virtual bool setEmpty(vector<bool> *v)
    {
        for(int i=0; i < v->size();i++)
            (*v)[i]=false;
    }

    // v1 <- v1 or v2
    virtual bool unionVect(vector<bool> *v1,vector<bool> *v2)
    {
        for(int i=0; i < v1->size();i++)
            (*v1)[i]=(*v1)[i]|(*v2)[i];
    }


    virtual bool printVector(vector<bool> *v)
    {
        cout << "[";
        for(int i=0; i < v->size();i++)
            cout << " " <<(*v)[i];
        cout << "]" << endl;

    }

    // Go through each line and if value matches a point print out the values
    virtual bool printReachDefResults()
    {
    }

    virtual bool runOnModule(Module& M)
    {


        for (Module::iterator MI = M.begin(), ME = M.end(); MI != ME; ++MI)
            runOnFunction(*MI);


        return false;
    }
};

char ReachingDef::ID = 0;
RegisterPass<ReachingDef> X("reach", "15745: Iterative Reaching Definition Analysis");

}
```