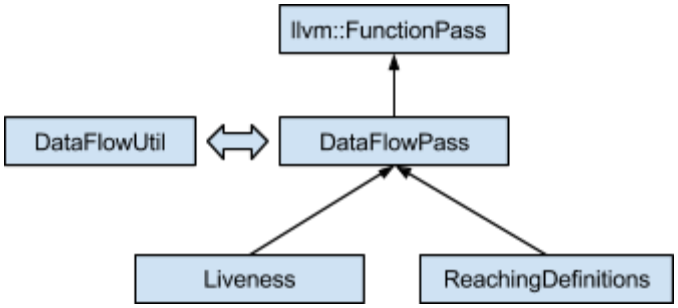
15-745 Homework 2 Writeup
Aditya Bhandaru, Zhe Qian

Framework Design
We take inheritance based approach in designing this data flow analysis framework.



The diagram above shows the class inheritance hierarchy. The `DataFlowPass` class derives directly from the LLVM `FunctionPass` class. In addition, it also uses static functions from the `DataFlowUtil` class (for things like set operations, etc.).

Leaf node classes such as Liveness and `ReachingDefinitions` extend the `DataFlowPass` class and call the super constructor with configuration parameters. These include the meet operator, direction of traversal, and the type of top set. Furthermore, subclasses of `DataFlowPass` must override a number of functions: `generate()`, `kill()`, and `transferFn()`. These are just mechanisms for setting the functional specifications of the desired data flow analysis.

The heart of the framework implementation lies within the DataFlowPass class. It overrides the runOnFunction method in the FunctionPass class, has logic for traversing the control flow graph, and performs the analysis specified the subclass. Finally, the DataFlowPass class contains logic for printing out the results of the analysis. For now, subclasses actually have very little logic in them. They could get bulkier as complexity increases.

Finally, all business logic and class definition code was split into header and implementation files for better style and cleaner reading.

Framework Testing
Using the test input supplied in previous assignments, we drew out the expected control flow graphs the the correct liveness and reaching definition outputs. We then matched these up with the program outputs for verification. The results for reaching definitions were fairly boring because the LLVM IR code was still in SSA notation.

Framework API

Step 1: To write your own pass, you should create a new class that extends `DataFlowPass`. The constructor must invoke the `DataFlowPass` constructor on instantiation with the correct configuration variables. You can find the enum definitions in `dataflow.h`. The body of your constructor may be empty. For example:

```
#include "dataflow.h"
using namespace llvm;
class ExampleAnalysis : public DataFlowPass {
  static char ID;
  ExampleAnalysis() : DataFlowPass(ID, NONE, UNION, BACKWARDS) { };
  // …
}
```

Step 2. You must override `generate()`, `kill()`, and `transferFn()` methods specified in `dataflow.h`. The generate and kill functions return a set of `Assignment` values (you can think of these as separate symbols) that a given block should generate and kill as the traversal is performed. The `transferFn()` method is the transfer function to apply on a block's inputs to get the outputs. You can find the class definition for `Assignment` in `util.h`. For example, your implementation may look like this:

```
class ExampleAnalysis : public DataFlowPass {
  // … constructor (above)
  Assignments Liveness::generate(const BasicBlock& block) {
    return DataFlowUtil::uses(block);
  }
  Assignments kill(const BasicBlock& block) {
    return DataFlowUtil::defines(block);
  }
  void transferFn(const Assignments& generate,
      const Assignments& kill, const Assignments& input, Assignments& output) {
    output = input;
    DataFlowUtil::setSubtract(output, kill);
    DataFlowUtil::setUnion(output, generate);
  }
};
```

Step 3. Finally, at the end of your implementation file, give your pass an ID and register it with LLVM separately.

```
char ExampleAnalysis::ID = 0;
RegisterPass<ExampleAnalysis> X("example", "15745 example");
```

Code Listing: dataflow.h, dataflow.cpp

```
// 15-745 S14 Assignment 2: dataflow.h
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////

#ifndef __CLASSICAL_DATAFLOW_DATAFLOW_H__
#define __CLASSICAL_DATAFLOW_DATAFLOW_H__

#include <iostream>
#include <queue>
#include <set>

#include "llvm/IR/Instructions.h"
#include "llvm/ADT/BitVector.h"
#include "llvm/ADT/DenseMap.h"
#include "llvm/ADT/DepthFirstIterator.h"
#include "llvm/ADT/SmallSet.h"
#include "llvm/ADT/ValueMap.h"
#include "llvm/Support/CFG.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Pass.h"

#include "util.h"


namespace llvm {


enum Meet {
  INTERSECTION,
  UNION
};

enum Direction {
  FORWARDS,
  BACKWARDS
};

enum Top {
  ALL,
  NONE
};

class DataFlowPass : public FunctionPass {
 public:
  DataFlowPass(char id, Top top, Meet meet, Direction direction);
  void computeGenKill(const Function& fn, BlockStates& states);
  void traverseForwards(const Function& fn, BlockStates& states);
  void traverseBackwards(const Function& fn, BlockStates& states);
  void meetFunction(const Assignments& in, Assignments& out);
  Assignments getTop(const Function& fn);
  void display(const Function& fn, BlockStates& states);
  // data flow API
  virtual Assignments generate(const BasicBlock& block) = 0;
  virtual Assignments kill(const BasicBlock& block) = 0;
  virtual void transferFn(const Assignments& generate, const Assignments& kill,
      const Assignments& input, Assignments& output) = 0;
```

```
  // pass API
  virtual bool runOnFunction(Function& F);
  virtual void getAnalysisUsage(AnalysisUsage& AU) const;

 protected:
  const Top _top;
  const Meet _meet;
  const Direction _direction;
};


}

#endif
// 15-745 S14 Assignment 2: dataflow.cpp
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////

#include "dataflow.h"

using std::cout;
using std::endl;


namespace llvm {


//
// Subclasses must call this constructor during construction.
// This constructor calls the FunctionPass constructor (it extends that class).
// Use initializer listed to assign constants on instantiation.
//
DataFlowPass::DataFlowPass(char id, Top top, Meet meet, Direction direction) :
    FunctionPass(id),
    _top(top),
    _meet(meet),
    _direction(direction) { };


//
// Compute the generate and kill sets for each basic block in the given
// function. The generate and kill functions are overriden by the subclass.
//
void DataFlowPass::computeGenKill(const Function& fn, BlockStates& states) {
  for (Function::const_iterator FI = fn.begin(), FE = fn.end(); FI != FE; ++FI) {
    const BasicBlock& block = *FI;
    BlockState state;
    state.generates = generate(block);
    state.kills = kill(block);
    // insert into map
    states.insert(BlockStatePair(&block, state));
  }
}


//
// Do a forwards traversal on the Control Flow Graph to perform the given
```

```
// analysis. The BlockState map is updated during the traversal.
//
void DataFlowPass::traverseForwards(const Function& fn, BlockStates& states) {
  std::queue<const BasicBlock*> worklist;
  std::set<const BasicBlock*> visited;

  // Set up initial conditions.
  Assignments top = getTop(fn);
  const BasicBlock* start = &(fn.front());
  BlockState& start_state = states[start];
  start_state.out = top;
  worklist.push(start);

  // Process queue until it is empty.
  while (!worklist.empty()) {
    // inspect 1st element
    const BasicBlock* current = worklist.front();
    worklist.pop();

    // determine the meet of all successors
    Assignments meet = top;
    for (const_pred_iterator I = pred_begin(current), IE = pred_end(current);
        I != IE; ++I) {
      BlockState& pred_state = states[*I];
      meetFunction(pred_state.out, meet);
    }

    // see if we need to inspect this node
    BlockState& state = states[current];
    if (visited.count(current) && DataFlowUtil::setEquals(state.in, meet)) {
      continue;
    }

    // perform transfer function
    state.in = meet;
    visited.insert(current);
    transferFn(state.generates, state.kills, state.in, state.out);

    // Add all predecessors to the worklist.
    for (succ_const_iterator I = succ_begin(current), IE = succ_end(current);
        I != IE; ++I) {
      worklist.push(*I);
    }
  }
}


//
// Do a backwards traversal on the Control Flow Graph to perform the given
// analysis. The BlockState map is updated during the traversal.
//
void DataFlowPass::traverseBackwards(const Function& fn, BlockStates& states) {
  std::queue<const BasicBlock*> worklist;
  std::set<const BasicBlock*> visited;

  // Set up initial conditions.
  Assignments top = getTop(fn);
```

```
  const BasicBlock* start = &(fn.back());
  BlockState& start_state = states[start];
  start_state.out = top;
  worklist.push(start);

  // Process queue until it is empty.
  while (!worklist.empty()) {
    // inspect 1st element
    const BasicBlock* current = worklist.front();
    worklist.pop();

    // determine the meet of all successors
    Assignments meet = top;
    for (succ_const_iterator I = succ_begin(current), IE = succ_end(current);
        I != IE; ++I) {
      BlockState& succ_state = states[*I];
      meetFunction(succ_state.in, meet);
    }

    // See if we need to inspect this node.
    BlockState& state = states[current];
    if (visited.count(current) && DataFlowUtil::setEquals(state.in, meet)) {
      // cout << "      <input unchanged>" << endl;
      continue;
    }

    // Perform transfer function.
    state.out = meet;
    visited.insert(current);
    transferFn(state.generates, state.kills, state.out, state.in);

    // Add all predecessors to the worklist.
    for (const_pred_iterator I = pred_begin(current), IE = pred_end(current);
        I != IE; ++I) {
      worklist.push(*I);
    }
  }
}


//
// Performs the correct meet operation on two input sets.
// The output is stored in the second set. The first set is unmodified.
//
void DataFlowPass::meetFunction(const Assignments& in, Assignments& out) {
  if (_meet == UNION) {
    DataFlowUtil::setUnion(out, in);
  } else if (_meet == INTERSECTION) {
    DataFlowUtil::setIntersect(out, in);
  }
}


//
// Returns the correct Top set based on the subclass configuration.
//
Assignments DataFlowPass::getTop(const Function& fn) {
```

```
  if (_top == ALL) {
    return DataFlowUtil::all(fn);
  } else {
    return Assignments();
  }
}


//
// Called after the pass is complete.
// Show the results of the pass for each program point b/t blocks.
//
void DataFlowPass::display(const Function& fn, BlockStates& states) {
  cout << "Function: " << fn.getName().data() << endl << endl;
  for (Function::const_iterator I = fn.begin(), IE = fn.end(); I != IE; ++I) {
    const BasicBlock* block = &(*I);
    BlockState& state = states[block];
    if (I == fn.begin()) {
      DataFlowUtil::print(state.in);
      cout << endl;
    }
    block->dump();
    DataFlowUtil::print(state.out);
    cout << endl;
  }
  cout << endl;
}


//
// Called by the FunctionPass API in LLVM.
//
bool DataFlowPass::runOnFunction(Function& fn) {
  // First pass: precompute generate and kill sets.
  BlockStates states;
  computeGenKill(fn, states);
  cout << endl;

  // iterate for a forwards pass
  if (_direction == FORWARDS) {
    traverseForwards(fn, states);
  }

  // iterate for a backwards pass
  else if (_direction == BACKWARDS) {
    traverseBackwards(fn, states);
  }

  // Does not modify the incoming Function.
  display(fn, states);
  return false;
}


void DataFlowPass::getAnalysisUsage(AnalysisUsage& AU) const {
  AU.setPreservesCFG();
}
```

```
}
```

Code Listing: utils.h, utils.cpp

```
// 15-745 S14 Assignment 2: util.h
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////

#ifndef __UTIL_H__
#define __UTIL_H__

#include <iostream>
#include <set>
#include <utility>

#include "llvm/ADT/DenseMap.h"
#include "llvm/ADT/SmallSet.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/Instructions.h"


namespace llvm {


//
// Useful types
//

class Assignment {
 public:
  Assignment(const Value *ptr) : pointer(ptr) { };
  const Value* pointer;
  inline bool operator==(const Assignment& rhs) const {
    return pointer == rhs.pointer;
  }
  inline bool operator<(const Assignment& rhs) const {
    return pointer < rhs.pointer;
  }
};

typedef std::set<Assignment> Assignments;

class BlockState {
 public:
  Assignments in;
  Assignments out;
  Assignments generates;
  Assignments kills;
};

typedef DenseMap<const BasicBlock*, BlockState> BlockStates;
typedef std::pair<const BasicBlock*, BlockState> BlockStatePair;


//
// Static library functions (for reuse)
```

```
//

class DataFlowUtil {
 public:
  static Assignments uses(const BasicBlock& block);
  static Assignments defines(const BasicBlock& block);
  static Assignments kills(const BasicBlock& block);
  static Assignments all(const Function& fn);
  static void setSubtract(Assignments& dest, const Assignments& src);
  static void setUnion(Assignments& dest, const Assignments& src);
  static void setIntersect(Assignments& dest, const Assignments& src);
  static bool setEquals(const Assignments& a, const Assignments& b);
  static void print(const Assignments& assignments);
};

}

#endif
// 15-745 S14 Assignment 2: util.cpp
// Group: akbhanda, zheq
//
// Overview: This file contains useful typedefs and utility functions for the
//   the DataFlowPass class and its subclasses. The DataFlowUtil class is
//   completely stateless.
////////////////////////////////////////////////////////////////////////////


#include "util.h"


using std::cout;
using std::endl;


namespace llvm {


//
// For a given BasicBlock, compute which variables are used.
//
Assignments DataFlowUtil::uses(const BasicBlock& block) {
  Assignments useSet;
  for (BasicBlock::const_reverse_iterator it = block.rbegin();
      it != block.rend(); ++it) {
    const Instruction& instr = *it;
    const User* user = &instr;
    // iterate through all operands
    User::const_op_iterator OI, OE;
    for(OI = user->op_begin(), OE = user->op_end(); OI != OE; ++OI) {
      Value* val = *OI;
      // check if the operand is used
      if (isa<Instruction>(val) || isa<Argument>(val)) {
        useSet.insert(Assignment(val));
      }
    }
    useSet.erase(Assignment(&instr));
  }
```

```
  return useSet;
}


//
// For a given BasicBlock, compute which variables are defined.
//
Assignments DataFlowUtil::defines(const BasicBlock& block) {
  Assignments defSet;
  for (BasicBlock::const_iterator it = block.begin(); it != block.end(); ++it) {
    // there's no result area for an instr, every instruction is actually a definition
    const Instruction& instr = *it;
    defSet.insert(Assignment(&instr));
  }
  return defSet;
}


Assignments DataFlowUtil::kills(const BasicBlock& block) {
  Assignments killSet;
  const Function& function = *block.getParent();
  for (BasicBlock::const_iterator it = block.begin(); it != block.end(); ++it) {
    const Instruction& inst = *it;

    for(Function::const_iterator itrF = function.begin(); itrF != function.end(); ++itrF)
{
      const BasicBlock& bb = *itrF;
      if (&bb == &block) {
        for(BasicBlock::const_iterator itrB = bb.begin(); itrB != bb.end(); ++itrB) {
          const Instruction& instr = *itrB;
          if (&inst == &instr) {
            killSet.insert(Assignment(&instr));
          }
        }
      }
    }
  }
  return killSet;
}


Assignments DataFlowUtil::all(const Function& fn) {
  Assignments all;

  // Function arguments are values too.
  for (Function::const_arg_iterator I = fn.arg_begin(), IE = fn.arg_end();
      I != IE; ++I) {
    const Argument* arg = &(*I);
    Assignment assign(arg);
    all.insert(assign);
  }

  // Populate with all instructions.
  // TODO: What do we do with loads and stores? Treat the same for now.
  for (Function::const_iterator I = fn.begin(), IE = fn.end(); I != IE; ++I) {
    const BasicBlock& block = *I;
    for (BasicBlock::const_iterator J = block.begin(), JE = block.end();
```

```
        J != JE; ++J) {
      const Instruction* instr = &(*J);
      Assignment assign(instr);
      all.insert(assign);
    }
  }
  return all;
}


//
// The following functions perform basic set operations in O(n*log(m)) time,
// where m and n are the sizes of the sets. For our purposes, this is fast
// enough.
//
// The result of these operations is stored back into the 1st argument.
//
void DataFlowUtil::setSubtract(Assignments& dest, const Assignments& src) {
  for (Assignments::const_iterator i = src.begin(); i != src.end(); ++i) {
    const Assignment& sub = *i;
    dest.erase(sub);
  }
}


void DataFlowUtil::setUnion(Assignments& dest, const Assignments& src) {
  for (Assignments::const_iterator i = src.begin(); i != src.end(); ++i) {
    const Assignment& add = *i;
    dest.insert(add);
  }
}


void DataFlowUtil::setIntersect(Assignments& dest, const Assignments& src) {
  for (Assignments::const_iterator i = src.begin(); i != src.end(); ++i) {
    const Assignment& test = *i;
    if (dest.count(test) < 1) {
      dest.erase(test);
    }
  }
}


//
// Determine if 2 sets contain the same elements.
//
bool DataFlowUtil::setEquals(const Assignments& a, const Assignments& b) {
  // make sure sets are the same length
  if (a.size() != b.size()) {
    return false;
  }

  // ensure they contain the same elements
  for (Assignments::const_iterator i = a.begin(); i != a.end(); ++i) {
    const Assignment& test = *i;
    if (b.count(test) < 1) {
      return false;
```

```
    }
  }
  return true;
}


//
// Helper/debug function to quickly print out a set of assignments.
// Later if we change how we implement assignment sets, we will have to update
// this function.
//
void DataFlowUtil::print(const Assignments& assignments) {
  cout << "{ ";
  for (Assignments::const_iterator I = assignments.begin(),
       IE = assignments.end(); I != IE; ++I) {
    cout << (*I).pointer->getName().data() << " ";
  }
  cout << "}";
}

}
```

Code Listing: liveness.h, liveness.cpp

```
// 15-745 S14 Assignment 2: liveness.h
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////

#ifndef __LIVENESS_H__
#define __LIVENESS_H__

#include "llvm/IR/Function.h"
#include "llvm/Pass.h"

#include "dataflow.h"
#include "util.h"

namespace llvm {


class Liveness : public DataFlowPass {
 public:
  static char ID;
  Liveness();
  Assignments generate(const BasicBlock& block);
  Assignments kill(const BasicBlock& block);
  void transferFn(const Assignments& generate, const Assignments& kill,
    const Assignments& input, Assignments& output);
};


}

#endif
// 15-745 S14 Assignment 2: liveness.cpp
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////
```

```
#include "liveness.h"

using std::cout;
using std::endl;

namespace llvm {

Liveness::Liveness() : DataFlowPass(ID, NONE, UNION, BACKWARDS) { };

//
// Override generate function of DataFlowPass to use uses().
//
Assignments Liveness::generate(const BasicBlock& block) {
  return DataFlowUtil::uses(block);
}


//
// Override generate function of DataFlowPass to use defines().
// Notation and naming may change later.
//
Assignments Liveness::kill(const BasicBlock& block) {
  return DataFlowUtil::defines(block);
}


//
// Subclasses override the transfer function.
// More transparent way to provide function pointers.
//
void Liveness::transferFn(const Assignments& generate,
    const Assignments& kill, const Assignments& input, Assignments& output) {
  output = input;
  DataFlowUtil::setSubtract(output, kill);
  meetFunction(generate, output);
}


//
// Do the following to meet the FunctionPass API
//
char Liveness::ID = 0;
RegisterPass<Liveness> X("live", "15745 Liveness");

}
```

Code Listing: reaching-definitions.h, reaching-definitions.cpp

```
// 15-745 S14 Assignment 2: reaching-definitions.h
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////

#ifndef __REACHING_DEFINITIONS_H__
#define __REACHING_DEFINITIONS_H__

#include "llvm/IR/Function.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
```

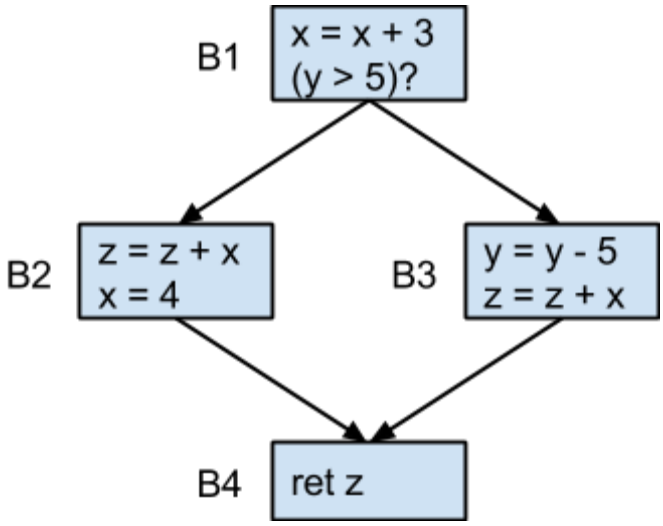
```
#include "dataflow.h"
#include "util.h"


namespace llvm {


class ReachingDefinitions : public DataFlowPass {
 public:
  static char ID;
  ReachingDefinitions();
  Assignments generate(const BasicBlock& block);
  Assignments kill(const BasicBlock& block);
  void transferFn(const Assignments& generate, const Assignments& kill,
    const Assignments& input, Assignments& output);
};


}

#endif
// 15-745 S14 Assignment 2: reaching-definitions.cpp
// Group: akbhanda, zheq
////////////////////////////////////////////////////////////////////////////////

#include "reaching-definitions.h"


using std::cout;
using std::endl;


namespace llvm {


ReachingDefinitions::ReachingDefinitions() :
    DataFlowPass(ID, NONE, UNION, FORWARDS) { };


//
// Override generate function of DataFlowPass to use defines().
//
Assignments ReachingDefinitions::generate(const BasicBlock& block) {
  return DataFlowUtil::defines(block);
}


//
// Override generate function of DataFlowPass to use kills().
// Notation and naming may change later.
//
Assignments ReachingDefinitions::kill(const BasicBlock& block) {
  return DataFlowUtil::kills(block);
}
```

```
//
// Subclasses override the transfer function.
// More transparent way to provide function pointers.
//
void ReachingDefinitions::transferFn(const Assignments& generate,
    const Assignments& kill, const Assignments& input, Assignments& output) {
  output = input;
  DataFlowUtil::setSubtract(output, kill);
  meetFunction(generate, output);
}


//
// Do the following to meet the FunctionPass API
//
char ReachingDefinitions::ID = 0;
RegisterPass<ReachingDefinitions> X("reach", "15745 ReachingDefinitions");


}
```

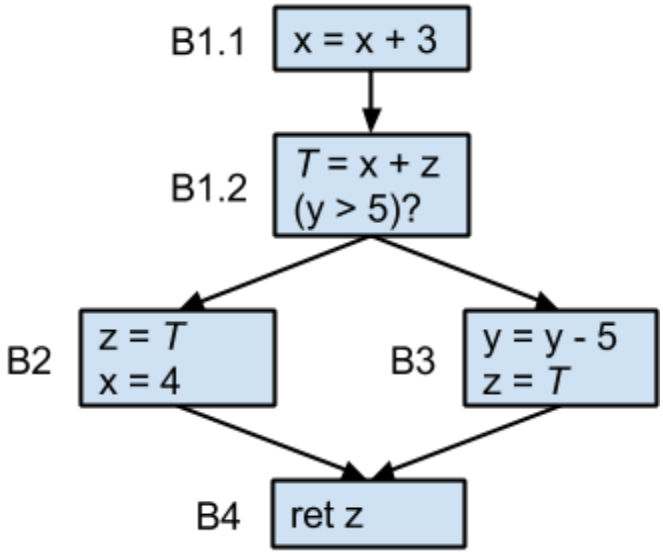
Theory Questions
Aditya Bhandaru (akbhanda), Zhe Qian (zheq)

3.1 Lazy Code Motion

3.1.1 Show the Control Flow Graph
The following is a CFG describing the code without any optimizations.



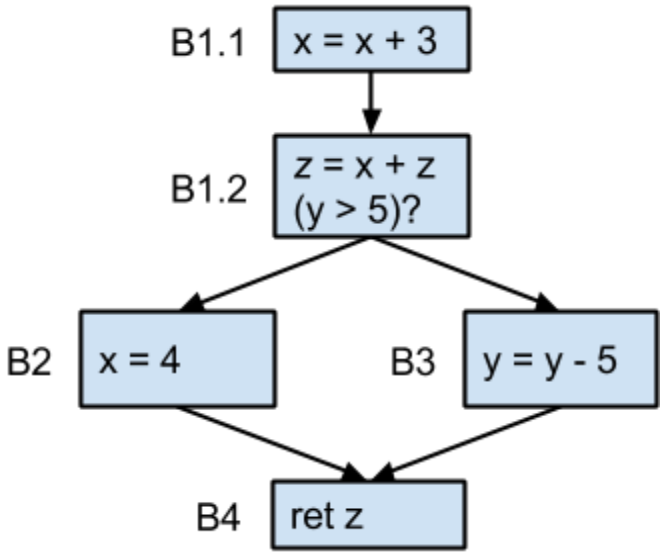
3.1.2 Show the graph after the Early Placement pass.
Here we show the modified graph, with the expression *z* + *x* computed after B1.



3.1.3 Show the graph after the LCM and Cleanup passes.
The computation of *x + z* cannot be delayed any further, as it is anticipated by both blocks B2 and B3. Therefore the graph does not change for the LCM pass.

For the cleanup pass, we remove dead assignments we create. Here, all paths go through block B4, therefore we know the expressions *x = 4* and *y = y - 5* are not anticipated anywhere else, and can be removed. We do not show that below, however.

Note: The Clean up pass may automatically remove the assignments to *x* and *y*, it depends on the implementation. For instance, you may only remove temporary variables you create.

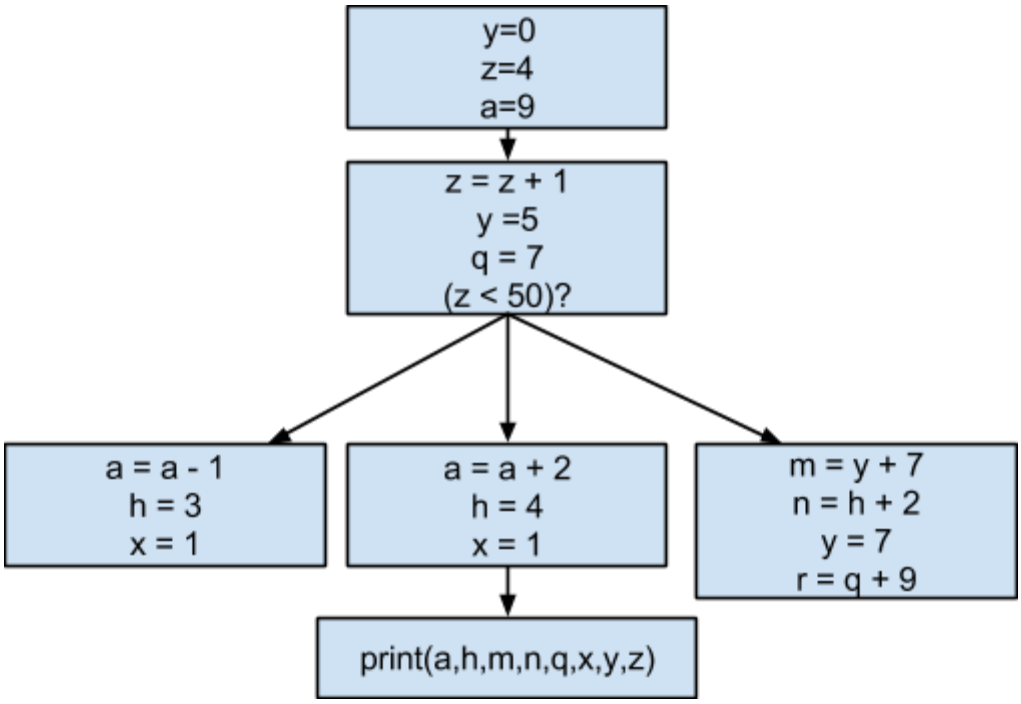


3.2 LICM: Loop Invariant Code Motion

3.2.1 Loop invariant instructions:

- x = 1;
- q = 7;
- r = q + 5; (easily discerned from value propagation and constant folding)
- y = 5; (y is constantly changing in the loop but y = 7 is actually dead code)
- m = y + 7; (this expression gets computed before y = 7 every time)

3.2.2 New dominator tree and reasoning.



Assignment *x = 1* and *q = 7* is clearly able to get moved to the pre-header. *q = 7* appears only once every repetition and never changed. *x = 1* is in both of two branches and the assigned value is same. Although *r = q +5* is also invariant but since its not used outside the loop, it should actually be eliminated, so there's no need to move it to the pre-header.

Assignment *y = 5* here is tricky. Indeed, another assignment *y = 7* exists in the loop and is executed every repetition. But it's actually dead code since that *y*'s value is reassigned right after that block and the value of 7 is never used. So *m = y + 7* is also invariant since y here is always 5.

In conclusion, all the loop invariant instructions except the *r = q + 5* should be moved to the pre-header.