# WRITING AN LLVM PASS

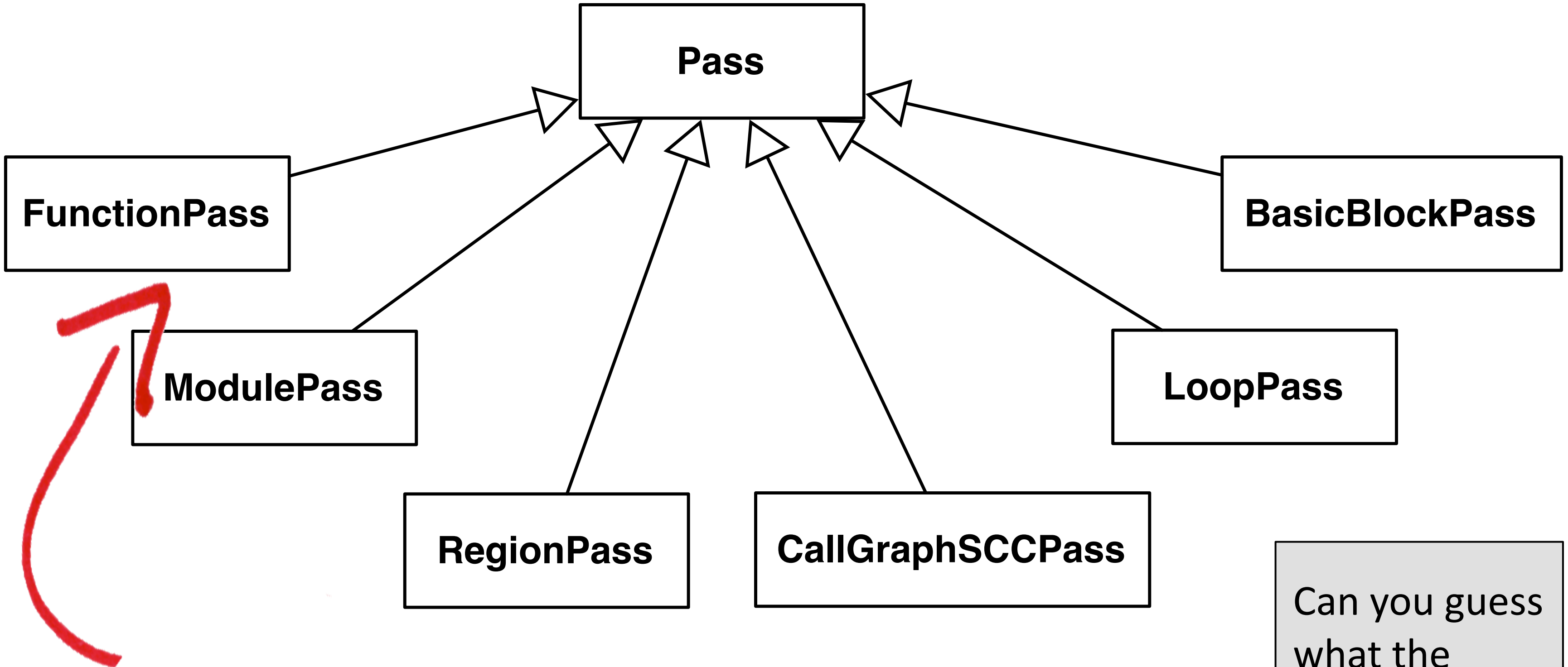
DCC 888

# Passes

- LLVM applies a chain of analyses and transformations on the target program.
- Each of these analyses or transformations is called a *pass*.
- We have seen a few passes already: `mem2reg`, `early-cse` and `constprop`, for instance.
- Some passes, which are machine independent, are invoked by opt.
- Other passes, which are machine dependent, are invoked by llc.
- A pass may require information provided by other passes. Such dependencies must be explicitly stated.
  - For instance: a common pattern is a transformation pass requiring an analysis pass.

# Different Types of Passes

- A pass is an instance of the LLVM class `Pass`.
- There are many kinds of passes.



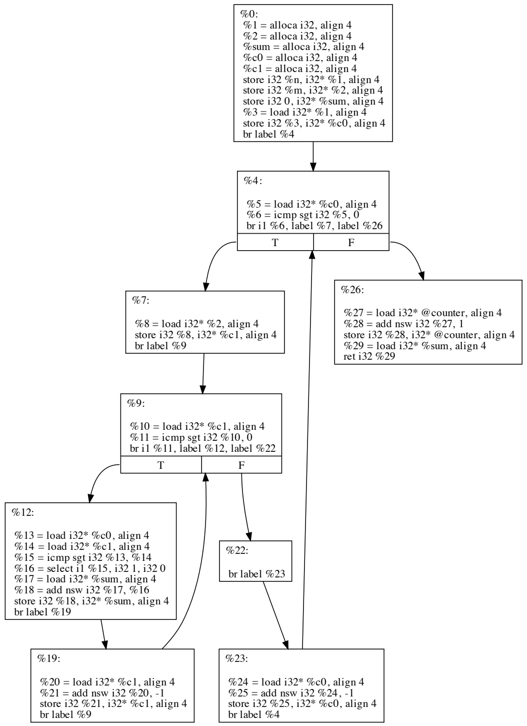

In this lesson we will focus on Function Passes, which analyze whole functions.

Can you guess what the other passes are good for?

# Counting Number of Opcodes in Programs

Let's write a pass that counts the number of times that each opcode appears in a given function. This pass must print, for each function, a list with all the instructions that showed up in its code, followed by the number of times each of these opcodes has been used.

```
int foo(int n, int m) {
  int sum = 0;
  int c0;
  for (c0 = n; c0 > 0; c0--) {
    int c1 = m;
    for (; c1 > 0; c1--) {
      sum += c0 > c1 ? 1 : 0;
    }
  }
  return sum;
}
```




```
Function foo
add: 4
alloca: 5
br: 8
icmp: 3
load: 11
ret: 1
select: 1
store: 9
```

# Counting Number of Opcodes in Programs

```
#define DEBUG_TYPE "opCounter"
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
#include <map>
using namespace llvm;
namespace {
  struct CountOp : public FunctionPass {
    std::map<std::string, int> opCounter;
    static char ID;
    CountOp() : FunctionPass(ID) {}
    virtual bool runOnFunction(Function &F) {
      errs() << "Function " << F.getName() << '\n';
      for (Function::iterator bb = F.begin(), e = F.end(); bb != e; ++bb) {
        for (BasicBlock::iterator i = bb->begin(), e = bb->end(); i != e; ++i) {
          if(opCounter.find(i->getOpcodeName()) == opCounter.end()) {
            opCounter[i->getOpcodeName()] = 1;
          } else {
            opCounter[i->getOpcodeName()] += 1;
          }
        }
      }
      std::map <std::string, int>::iterator i = opCounter.begin();
      std::map <std::string, int>::iterator e = opCounter.end();
      while (i != e) {
        errs() << i->first << ": " << i->second << "\n";
        i++;
      }
      errs() << "\n";
      opCounter.clear();
      return false;
    }
  };
}
char CountOp::ID = 0;
static RegisterPass<CountOp> X("opCounter", "Counts opcodes per functions");
```

Our pass runs once for each function in the program; therefore, it is a `FunctionPass`. If we had to see the whole program, then we would implement a `ModulePass`.

What are anonymous namespaces?

**This** line defines the name of the pass, in the command line, e.g., opCounter, and the help string that opt provides to the user about the pass.

# A Closer Look into our Pass

```
struct CountOp : public FunctionPass {
   std::map<std::string, int> opCounter;
   static char ID;
   CountOp() : FunctionPass(ID) {}
   virtual bool runOnFunction(Function &F) {
     errs() << "Function " << F.getName() << '\n';
     for (Function::iterator bb = F.begin(), e = F.end(); bb != e; ++bb) {
       for (BasicBlock::iterator i = bb->begin(), e = bb->end(); i != e; ++i) {
         if(opCounter.find(i->getOpcodeName()) == opCounter.end()) {
           opCounter[i->getOpcodeName()] = 1;
         } else {
           opCounter[i->getOpcodeName()] += 1;
         }
       }
     }
     std::map <std::string, int>::iterator i = opCounter.begin();
     std::map <std::string, int>::iterator e = opCounter.end();
     while (i != e) {
       errs() << i->first << ": " << i->second << "\n";
       i++;
     }
     errs() << "\n";
     opCounter.clear();
     return false;
   }
  };
}
```

We will be recording the number of each opcode in **this** map, that binds opcode names to integer numbers.

**This** code collects the opcodes. We will look into it more closely soon.

**This** code prints our results. It is a standard loop on an STL data structure. We use iterators to go over the map. Each element in a map is a pair, where the first element is the key, and the second is the value.

# Iterating Through Functions, Blocks and Insts

```
for(Function::iterator bb = F.begin(), e = F.end(); bb != e; ++bb) {
  for(BasicBlock::iterator i = bb->begin(), e = bb->end(); i != e; ++i) {
    if(opCounter.find(i->getOpcodeName()) == opCounter.end()) {
      opCounter[i->getOpcodeName()] = 1;
    } else {
      opCounter[i->getOpcodeName()] += 1;
    }
  }
}
```

We go over LLVM data structures through iterators.

- An **iterator over a Module** gives us a list of Functions.
- An **iterator over a Function** gives us a list of basic blocks.
- An **iterator over a Block** gives us a list of instructions.
- And we can **iterate over the operands** of the instruction too.

```
for (Module::iterator F = M.begin(), E = M.end(); F != E; ++F);

for (User::op_iterator O = I.op_begin(), E = I.op_end(); O != E; ++O);
```

# Compiling the Pass

- To Compile the pass, we can follow these two steps:

  1. We may save the pass into♡ `llvm/lib/Transforms/DirectoryName`, where `DirectoryName` can be, for instance, `CountOp`.

  2. We build a Makefile for the project. If we invoke the LLVM standard Makefile, we save some time.

```
# Path to top level of LLVM hierarchy
LEVEL = ../../..

# Name of the library to build
LIBRARYNAME = CountOp

# Make the shared library become a
# loadable module so the tools can
# dlopen/dlsym on the resulting library.
LOADABLE_MODULE = 1

# Include the makefile implementation
include $(LEVEL)/Makefile.common
```

♡: Well, given that this pass does not change the source program, we could save it in the Analyses folder. For more info on the LLVM structure, see http://llvm.org/docs/Projects.html

# Running the Pass

- Our pass is now a shared library, in `llvm/Debug/lib`[1].
- We can invoke it using the opt tool:

Just to avoid printing the binary t.bc file

```
$> clang –c –emit-llvm file.c –o file.bc

$> opt -load CountOp.dylib -opCounter -disable-output t.bc
```

- Remember, if we are running on Linux, then our shared library has the extension "`.so`", instead of "`.dylib`", as in the Mac OS.

[1]: Actually, the true location of the new library depends on your system setup. If you have compiled LLVM with the –Debug directive, for instance, then your binaries will be in llvm/Release/lib.

# Registering the Pass

The command *static RegisterPass<CountOp> X("opCounter", "Counts opcodes per functions");* registers the pass in the LLVM's pass manager:

```
$> opt -load CountOp.dylib -help

OVERVIEW: llvm .bc -> .bc modular optimizer and analysis printer

USAGE: opt [options] <input bitcode file>

OPTIONS:
  -O1                                - Optimization level 1.
  -O2                                - Optimization level 2.
  ...
  Optimizations available:
    ...
    -objc-arc-contract               - ObjC ARC contraction
    -objc-arc-expand                 - ObjC ARC expansion
    -opCounter                       - Counts opcodes per functions
    -partial-inliner                 - Partial Inliner
  ...
  -x86-asm-syntax                    - Choose style of code to emit:
    =att                             -   Emit AT&T-style assembly
    =intel                           -   Emit Intel-style assembly
```

# Timing the Pass

The pass manager provides the time-passes directive, that lets us get the runtime of each pass that we run. That is useful during benchmarking.

```
$> opt -load CountOp.dylib -opCounter -disable-output -time-passes f.bc

Function main
add: 6
br: 17
call: 1
icmp: 5
ret: 1

===-------------------------------------------------------------------------===
                      ... Pass execution timing report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0010 seconds (0.0011 wall clock)

   ---User Time---   --System Time--   --User+System--   ---Wall Time---  --- Name ---
   0.0002 ( 30.6%)   0.0002 ( 57.7%)   0.0004 ( 37.7%)   0.0004 ( 39.2%)  Counts opcodes
                                                                          per functions
   0.0003 ( 33.6%)   0.0001 ( 21.1%)   0.0003 ( 30.3%)   0.0003 ( 29.3%)  Module
                                                                          Verifier
   0.0003 ( 34.6%)   0.0001 ( 18.9%)   0.0003 ( 30.5%)   0.0003 ( 29.2%)  Dominator Tree
                                                                          Construction
   0.0000 (  1.2%)   0.0000 (  2.3%)   0.0000 (  1.5%)   0.0000 (  2.3%)  Preliminary
                                                                          verification
   0.0008 (100.0%)   0.0003 (100.0%)   0.0010 (100.0%)   0.0011 (100.0%)  Total
```

Can you guess what these other passes are doing?

# Chaining Passes

- A pass may invoke another.
  - To transform the program, e.g., BreakCriticalEdges
  - To obtain information about the program, e.g., AliasAnalysis
- If a pass invokes another, then it must say it explicitly, through the `getAnalysisUsage` method, in the class `FunctionPass`.
- To recover the data-structures created by the pass, we can use the `getAnalysis` method.

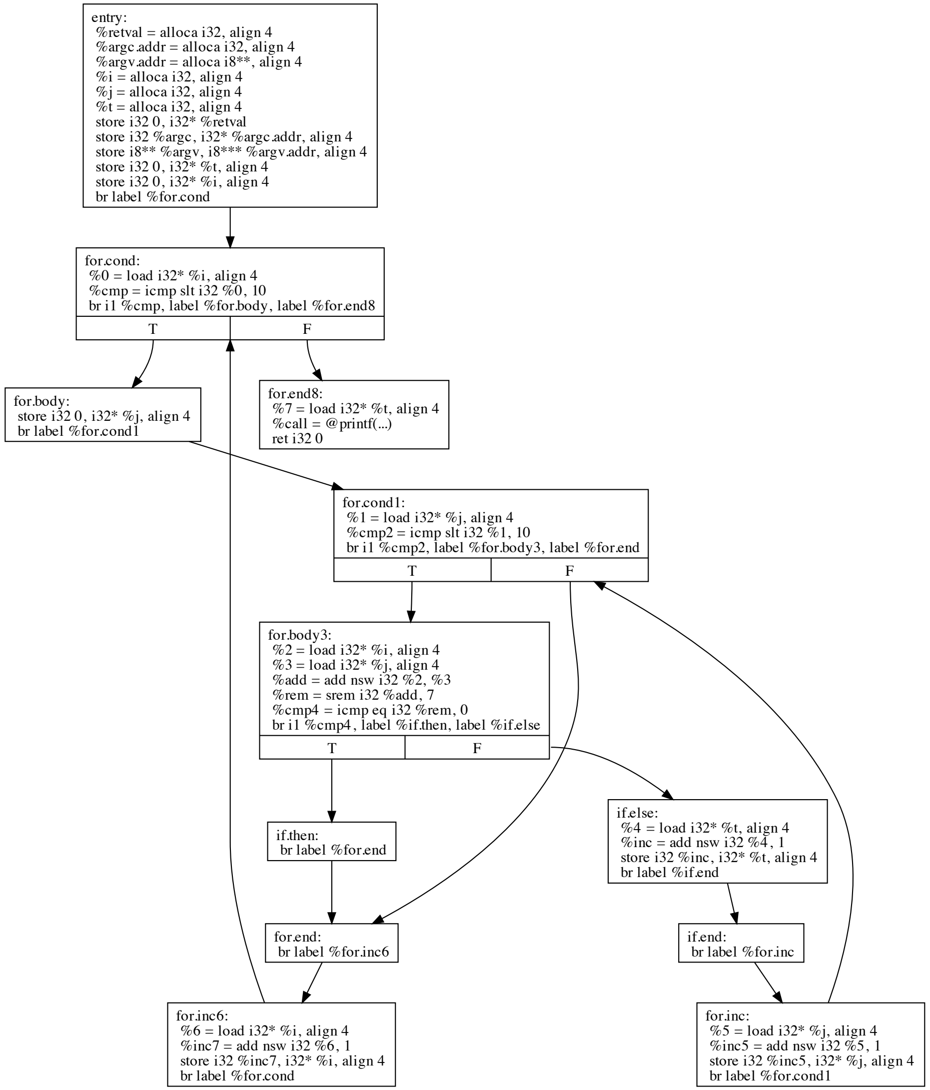
# Counting the Number of Basic Blocks in Loops

In order to demonstrate how to invoke a pass from another pass, we will create a tool to count the number of basic blocks inside a loop.

1) How many loops do we have in the program on the right?

2) How to identify a loop?

3) How many basic blocks do we have in the smallest loop?



# Dealing with Loops

In order to demonstrate how to invoke a pass from another pass, we will create a tool to count the number of basic blocks inside a loop.

1) How many loops do we have in the program on the right?
2) How to identify a loop?
3) How many basic blocks do we have in the smallest loop?

- We could implement some functionalities to deal with all the questions on the left.
- However, LLVM already has a pass that handles loops.
- We can use this pass to obtain the number of basic blocks per loops.

# The Skeleton of our Pass

```
namespace {
 struct BBinLoops : public FunctionPass {
  static char ID;
  BBinLoops() : FunctionPass(ID) {}

  void getAnalysisUsage(AnalysisUsage &AU) const {
   …
  }

  virtual bool runOnFunction(Function &F) {
   …
   return(false);
  }
 };
}

char BBinLoops::ID = 0;
static RegisterPass<BBinLoops> X("bbloop",
  "Count the number of BBs inside each loop");
```

What is the difference between structs and classes in C++?

1) We will be going over functions; hence, we implement a **FunctionPass**.
2) A pass, in LLVM, is implemented as a class (or a **struct**, as they are almost the same in C++).
3) **This method** tells LLVM which other passes we need to execute properly.
4) Our pass is not changing the program, thus we **return false**. Were we applying any change on the program, then our runOnFunction method should return true.

# Which Analyses do you Need?

- An LLVM pass must declare which other passes it requires to execute properly.
  - This declaration is done in the `getAnalysisUsage` method.

```
void getAnalysisUsage(AnalysisUsage &AU) const {
    AU.addRequired<LoopInfo>();
    AU.setPreservesAll ();
}
```

In our example, we are saying that LoopInfo – an LLVM pass – is required by our analysis. We are also saying that we do not change the program in any way that would invalidate the results computed by other passes. If another pass, later on, also requires LoopInfo, then the information stored in LoopInfo will not need to be recomputed, for instance.

# The Basic Block Counter

```
virtual bool runOnFunction(Function &F) {
   LoopInfo &LI = getAnalysis<LoopInfo>();
   int loopCounter = 0;
   errs() << F.getName() + "\n";
   for (LoopInfo::iterator i = LI.begin(), e = LI.end(); i != e; ++i) {
      Loop *L = *i;
      int bbCounter = 0;
      loopCounter++;
      for(Loop::block_iterator bb = L->block_begin(); bb != L->block_end(); ++bb) {
         bbCounter+=1;
      }
      errs() << "Loop ";
      errs() << loopCounter;
      errs() << ": #BBs = ";
      errs() << bbCounter;
      errs() << "\n";
   }
   return(false);
}
```

What do we get with these iterators?

We are using a data-structure called LoopInfo, which is produced by a pass with the same name. We obtain a pointer to this pass via get getAnalysis method, which is parameterized by a type, e.g., the class name LoopInfo.

# Using LoopInfo

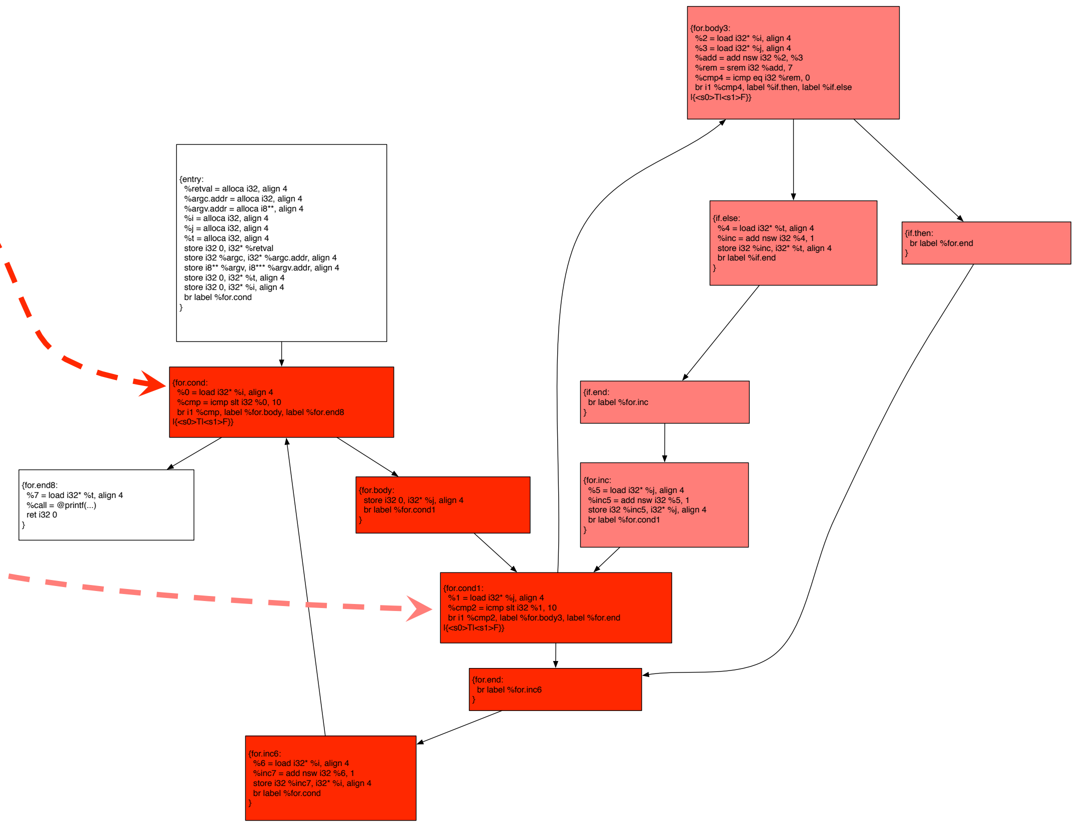
```
virtual bool runOnFunction(Function &F) {
  LoopInfo &LI = getAnalysis<LoopInfo>();
  int loopCounter = 0;
  errs() << F.getName() + "\n";
  for (LoopInfo::iterator i = LI.begin(), e = LI.end(); i != e; ++i) {
    Loop *L = *i;
    int bbCounter = 0;
    loopCounter++;
    for(Loop::block_iterator bb = L->block_begin(); bb != L->block_end(); ++bb) {
      bbCounter+=1;
    }
    errs() << "Loop ";
    errs() << loopCounter;
    errs() << ": #BBs = ";
    errs() << bbCounter;
    errs() << "\n";
  }
  return(false);
}
```

An iterator on LoopInfo gives us a collection of loops. An iterator on a Loop gives us a collection of basic blocks that constitute that loop.

# Iterating on Loops

```
for (LoopInfo::iterator i = LI.begin(), e = LI.end(); i != e; ++i) {
    ...
}
```

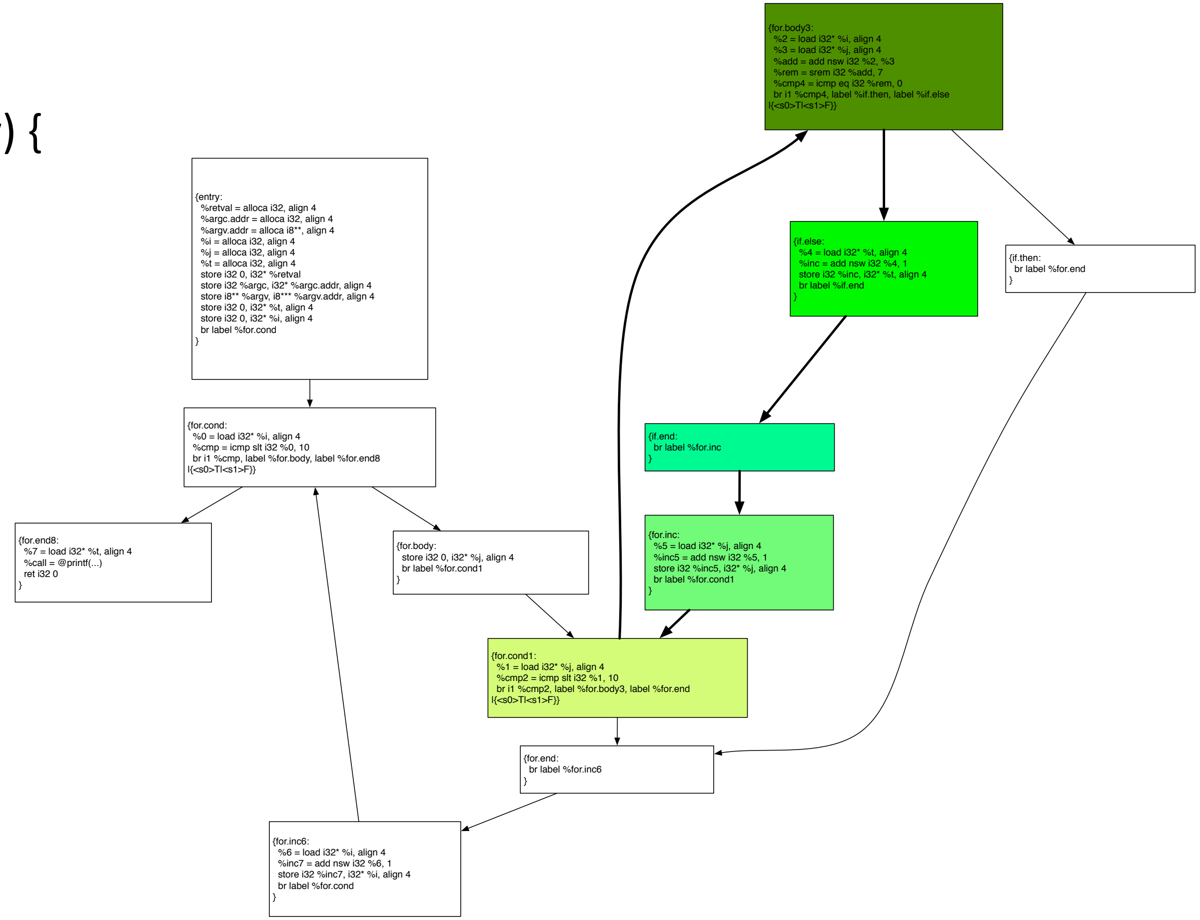
```
int main(int argc, char **argv) {
  int i, j, t = 0;
  for(i = 0; i < 10; i++) {
    for(j = 0; j < 10; j++) {
      if((i + j) % 7 == 0)
        break;
      else
        t++;
    }
  }
  printf("%d\n", t);
  return 0;
}
```



# Iterating on Blocks inside Loops

```
for (LoopInfo::iterator i = LI.begin(), e = LI.end(); i != e; ++i) {
  Loop *L = *i;
  for(Loop::block_iterator bb = L->block_begin(); bb != L->block_end(); ++bb) {}
}
```

```
int main(int argc, char **argv) {
 int i, j, t = 0;
 for(i = 0; i < 10; i++) {
  for(j = 0; j < 10; j++) {
   if((i + j) % 7 == 0)
    break;
   else
    t++;
  }
 }
 printf("%d\n", t);
 return 0;
}
```



# Running the Counter

- Again, once we compile this pass, we can invoke it using the opt tool, like we did before:

```
$> clang –c –emit-llvm file.c –o file.bc

$> opt -load dcc888.dylib -bbloop -disable-output file.bc
```

The results of our pass will be printed in the standard error output, as we are using the `errs()` channel to output results.

```
Function main
Loop 1: #BBs = 10
```

Ouf, now wait: we have two loops. What happened to the second one?

# Fixing the Loop Counter

```
virtual bool runOnFunction(Function &F) {
  LoopInfo &LI = getAnalysis<LoopInfo>();
  int loopCounter = 0;
  errs() << F.getName() + "\n";
  for (LoopInfo::iterator i = LI.begin(), e = LI.end(); i != e; ++i) {
    Loop *L = *i;
    int bbCounter = 0;
    loopCounter++;
    for(Loop::block_iterator bb = L->block_begin(); bb != L->block_end(); ++bb) {
      bbCounter+=1;
    }
    errs() << "Loop ";
    errs() << loopCounter;
    errs() << ": #BBs = ";
    errs() << bbCounter;
    errs() << "\n";
  }
  return(false);
}
```

Any idea on how could we fix it?

This code only goes over the outermost loops of a function. It does not really visit nested loops.

# Recursively Navigating Through Loops

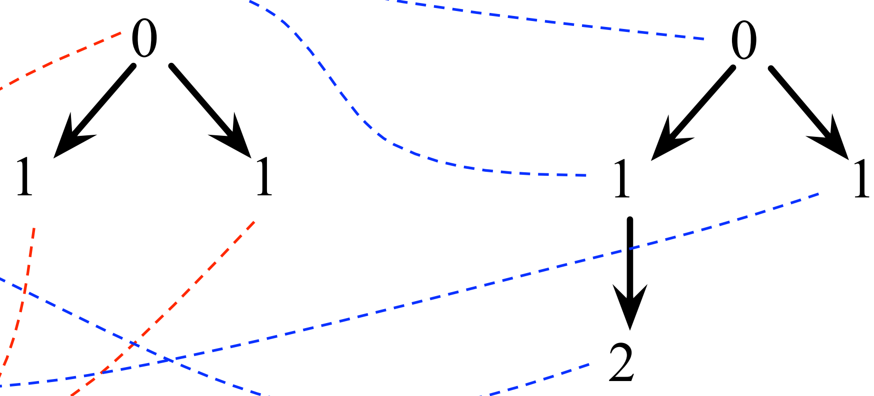
```
void countBlocksInLoop(Loop *L, unsigned nesting) {
  unsigned numBlocks = 0;
  Loop::block_iterator bb;
  for(bb = L->block_begin(); bb != L->block_end();++bb)
    numBlocks++;
  errs() << "Loop level " << nesting << " has " << numBlocks << " blocks\n";
  vector<Loop*> subLoops = L->getSubLoops();
  Loop::iterator j, f;
  for (j = subLoops.begin(), f = subLoops.end(); j != f; ++j)
    countBlocksInLoop(*j, nesting + 1);
}

virtual bool runOnFunction(Function &F) {
  LoopInfo &LI = getAnalysis<LoopInfo>();
  errs() << "Function " << F.getName() + "\n";
  for (LoopInfo::iterator i = LI.begin(), e = LI.end(); i != e; ++i)
    countBlocksInLoop(*i, 0);
  return(false);
}
```

We can use the getSubLoop method to obtain a handle for the nested loops.

Are you sure this recursion terminates?

# The Fix in Action

```
int main(int argc, char **argv) {
  int i, j, k, t = 0;
  for(i = 0; i < 10; i++) {
    for(j = 0; j < 10; j++) {
      for(k = 0; k < 10; k++) {
        t++;
      }
    }
    for(j = 0; j < 10; j++) {
      t++;
    }
  }
  for(i = 0; i < 20; i++) {
    for(j = 0; j < 20; j++) {
      t++;
    }
    for(j = 0; j < 20; j++) {
      t++;
    }
  }
  return t;
}
```




```
$> opt -load dcc888.dylib -bbloop -disable-output ex.bc
Function main
Loop level 0 has 11 blocks
Loop level 1 has 3 blocks
Loop level 1 has 3 blocks
Loop level 0 has 15 blocks
Loop level 1 has 7 blocks
Loop level 2 has 3 blocks
Loop level 1 has 3 blocks
```

# Which Passes do I Invoke?

- The LLVM's pass manager provides a debug-pass option that gives us the chance to see which passes interact with our analyses and optimizations:

```
$> opt -load dcc888.dylib -bbloop -disable-output --debug-pass=Structure file.bc
```

There are other options that we can use with debug-pass:
- Arguments
- Details
- Disabled
- Executions
- Structure

```
Target Library Information
Data Layout
No target information
Target independent code generator's TTI
X86 Target Transform Info
  ModulePass Manager
    FunctionPass Manager
      Dominator Tree Construction
      Natural Loop Information
      Count the number of BBs inside each loop
      Preliminary module verification
      Module Verifier
```

# Final Remarks

- LLVM provides users with a string of analyses and optimizations which are called passes.
- Users can chain new passes into this pipeline.
- The pass manager orders the passes in such a a way to satisfies the dependencies.
- Passes are organized according to their granularity, e.g., module, function, loop, basic block, etc.