Code analysis *and* transformation

LLVM

Simone Campanoni
simone.campanoni@northwestern.edu

Outline

- Introduction to LLVM
- Homework steps

- Hacking LLVM with CAT

LLVM

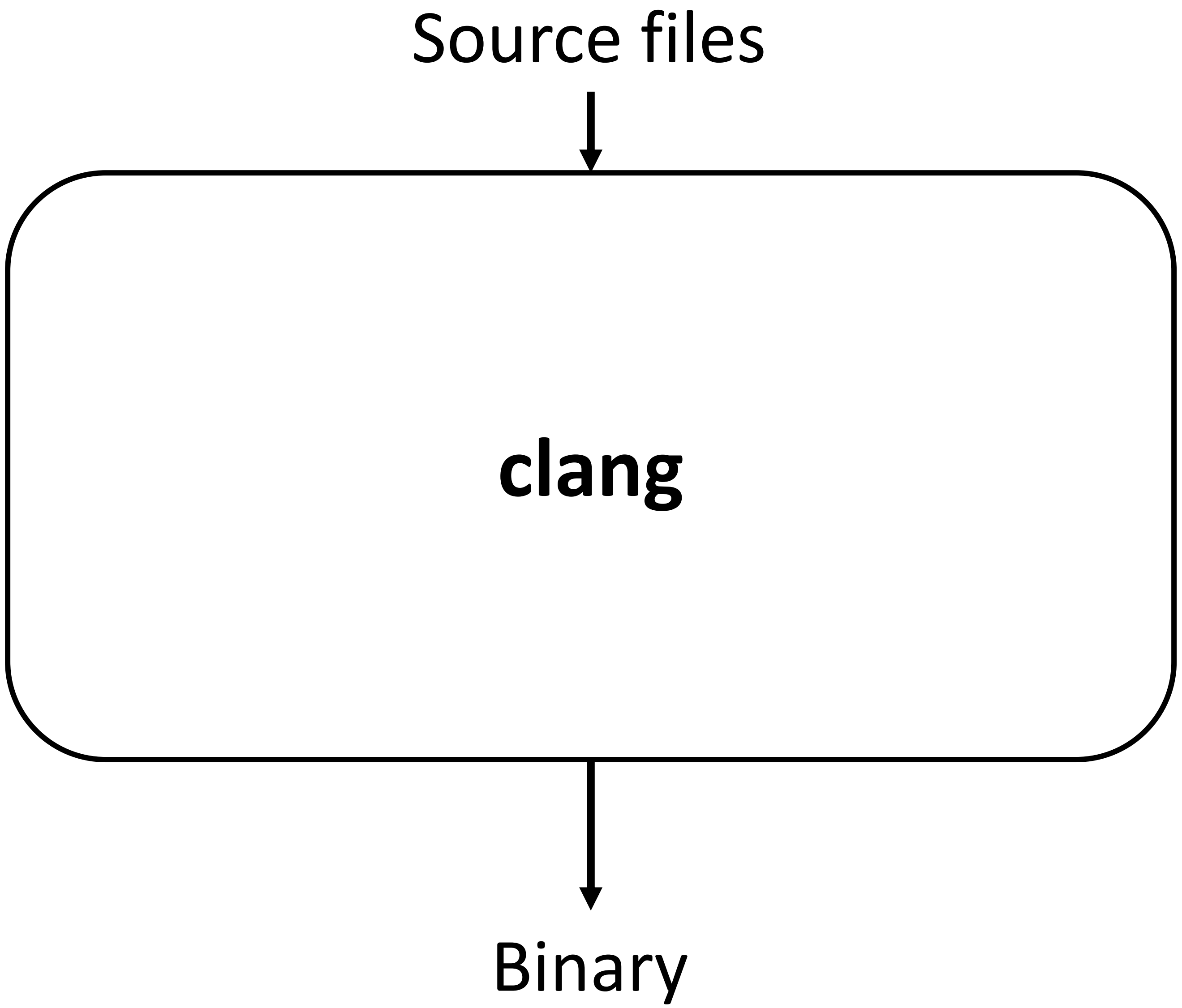
- LLVM is a great, hackable compilation framework
  - For C, C++, Objective-C, Swift, Rust, …
- But it's also (this is not a complete list)
  - A dynamic compiler
  - A compiler for bytecode languages (e.g., Java and CIL bytecode)
- LLVM IR
- LLVM is modular and well documented
- Started from UIUC, it's now the research tool of choice
- It's an industrial-strength codebase

  Apple, AMD, Intel, NVIDIA, …

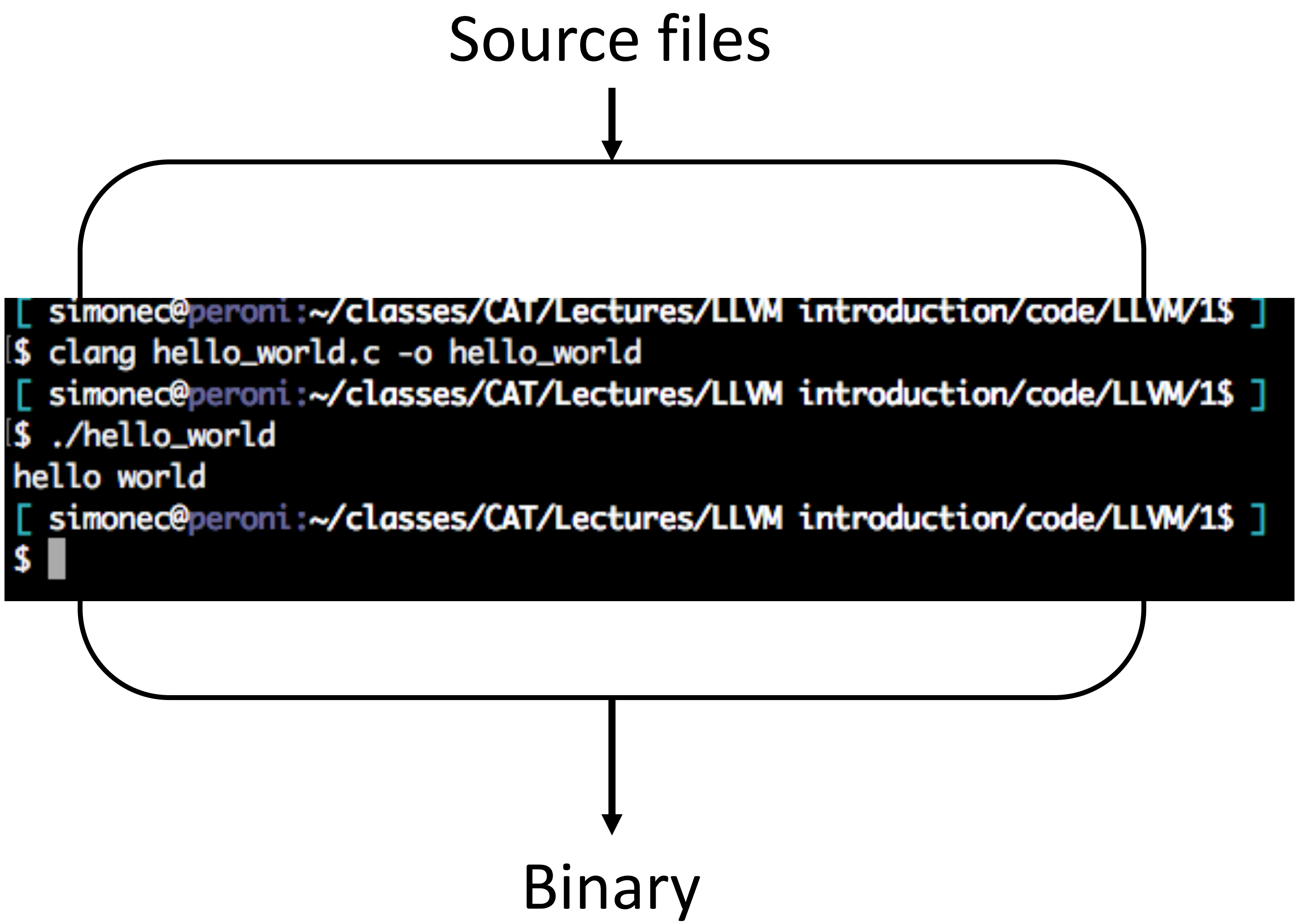
Tools built with LLVM

- clang: compile C/C++ code as well as OpenMP code
- clang-format: to format C/C++ code
- clang-tidy: to detect and fix bug-prone patterns, performance, portability and maintainability issues
- clangd: to make editors (e.g., vim) smart
- clang-rename: to refactor C/C++ code
- SAFECode: memory checker
- lldb: debugger
- lld: linker
- polly: parallelizing compiler for numerical and regular workloads (e.g., matrix multiplication)
- libclc: OpenCL standard library
- dragonegg: integrate GCC parsers
- vmkit: bytecode virtual machines
- … and many more

LLVM common use at 10000 feet

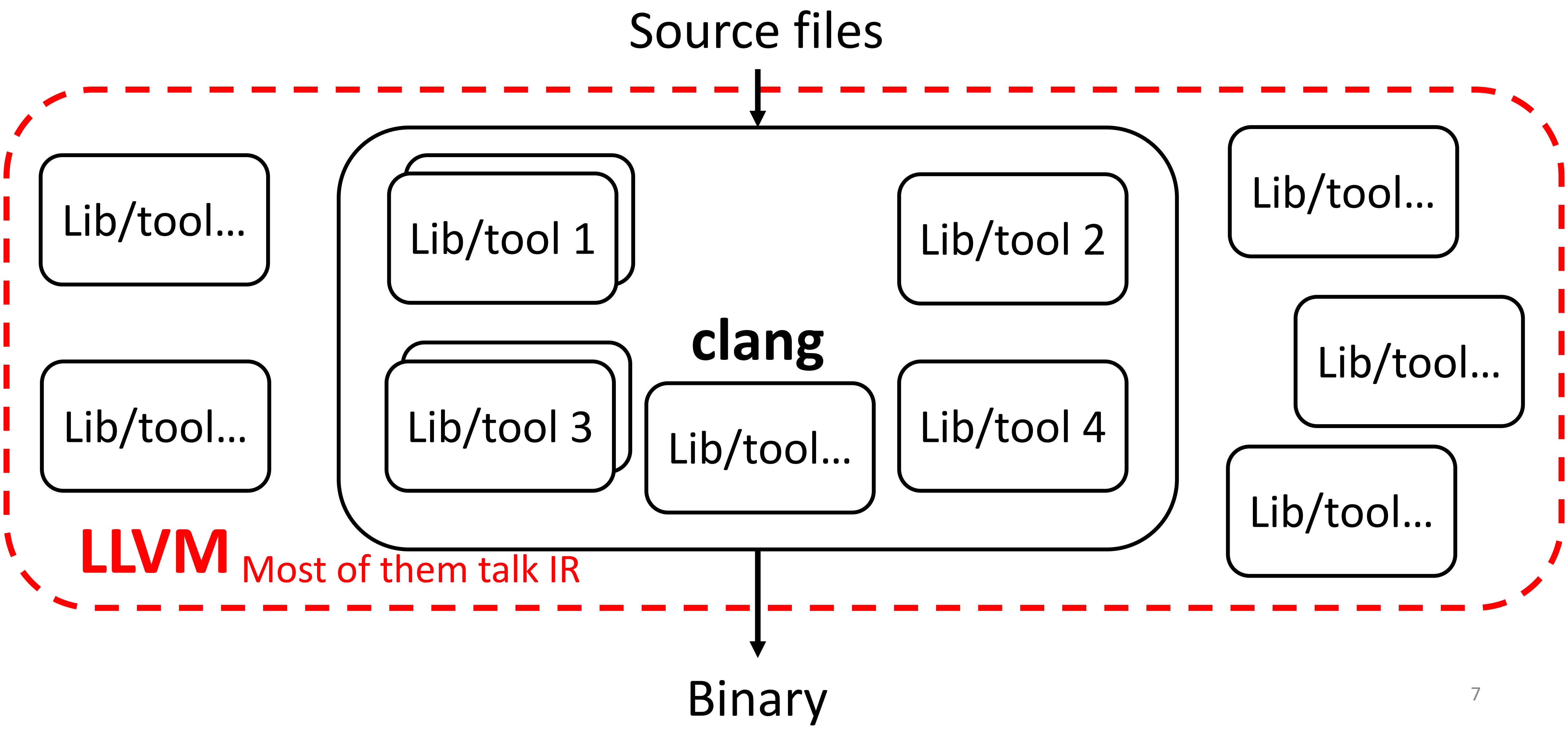


LLVM common use at 10000 feet



```
[ simonec@peroni:~/classes/CAT/Lectures/LLVM introduction/code/LLVM/1$ ]
$ clang hello_world.c -o hello_world
[ simonec@peroni:~/classes/CAT/Lectures/LLVM introduction/code/LLVM/1$ ]
$ ./hello_world
hello world
[ simonec@peroni:~/classes/CAT/Lectures/LLVM introduction/code/LLVM/1$ ]
$ 
```

LLVM common use at 10000 feet



LLVM internals

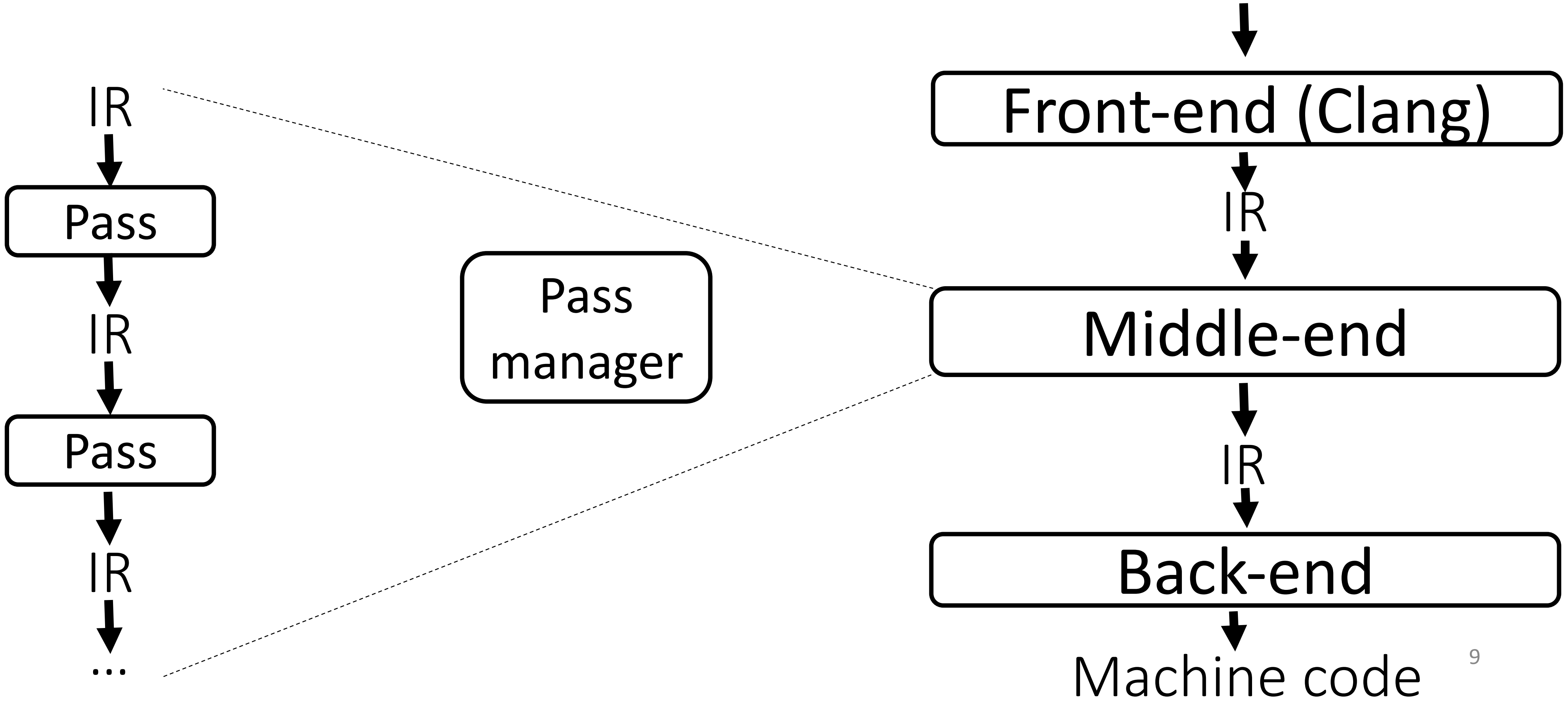
- An LLVM tool includes a compilation pipeline
    - Each stage: reads something as input and
      generates something as output
    - To develop a stage: specify how to transform the input
      to generate the output

- Most complexity in linking stages
  is kept outside the development of a stage

- In this class: we'll look at concepts and internals of middle-end
  But some of them are still valid for front-end/back-end

LLVM and other compilers

- LLVM middle-end is designed around it's IR



A middle-end pass in LLVM

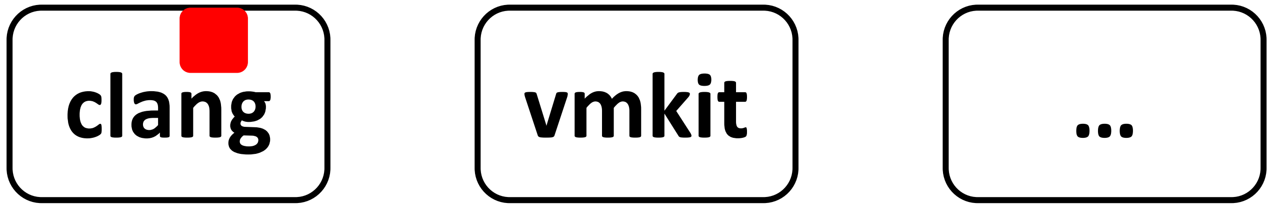
- A compilation pass reads
  and (sometime) modifies the bitcode (LLVM IR)

- If you want to analyze code:
  you need to understand the IR

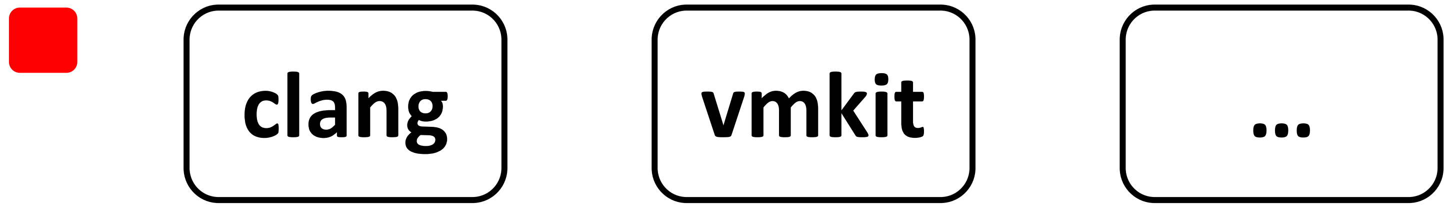
- If you want to modify the bitcode:
  you need to understand the IR

Adding a pass

- Internally



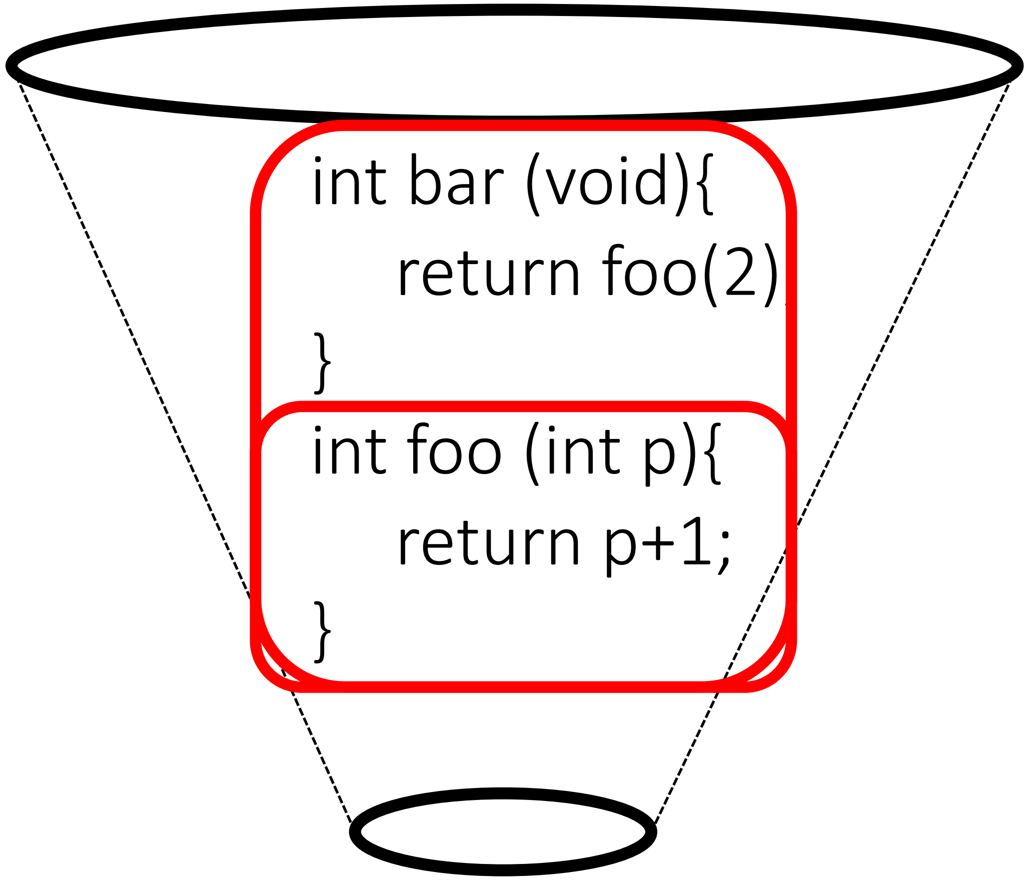
- Externally
  - More convenient to develop (compile-debug loop is much faster!)



Pass types

Use the "smallest" one for your CAT

- CallGraphSCCPass
- ModulePass
- FunctionPass
- LoopPass
- BasicBlockPass



```
int bar (void){
    return foo(2);
}
int foo (int p){
    return p+1;
}
```

Pass manager

- The pass manager orchestrates passes
- It builds the pipeline of passes in the middle-end
- The pipeline is created by respecting the dependences declared by each pass

  Pass X depends on Y
  Y will be invoked before X

Learning LLVM

- **Login (e.g.,** hanlon.wot.eecs.northwestern.edu**) and play with LLVM**
  - **LLVM 14.0.6 is installed in** /home/software/llvm
  - **Add the following code in both** ~/.bash_profile **and** ~/.bashrc **files**

```
LLVM_HOME=/home/software/llvm
export PATH=$LLVM_HOME/bin:$PATH
export LD_LIBRARY_PATH=$LLVM_HOME/lib:$LD_LIBRARY_PATH
```

- Get familiar with LLVM documentation
  - Doxygen pages (API docs)
  - Language reference manual (IR)
  - Programmer's manual (LLVM-specific data structures, tools)
  - Writing an LLVM pass

- Read the documentation
- Read the documentation

LLVM summary

- LLVM is an industrial-strength compiler
  also used in academia
  - Very hard to know in detail every component
  - Focus on what’s important for your goal
  - Become a ninja at jumping around the documentation
- It’s well organized, documented
  with a large community behind it
- Basic C++ skills are required

Final tips

- LLVM includes a LOT of passes
  - Analyses
  - Transformations
  - Normalization
- Take advantage of existing code
- I have a pointer to something. What is it?

  getName() works on most things

  errs() << TheThingYouDon’tKnow ;

Outline

- Introduction to LLVM
- Homework steps

- Hacking LLVM with CAT

Homework: build your own compiler

- You have a skeleton of a compiler (cat-c) built upon clang
  - https://github.com/scampanoni/LLVM_middleend_template
  - Switch to the branch v14: git checkout v14
  - This extends only the middle-end of clang by adding a new pass
  - This new pass will be invoked as last pass in the middle-end (independently whether you use O0, O1, O2, …)

- You will extend this skeleton to do all of your assignments

- You can only rely on what's included in LLVM (no external tools/analyses/transformations)

Homework: build your own compiler

To install cat-c (this needs to be done only once):

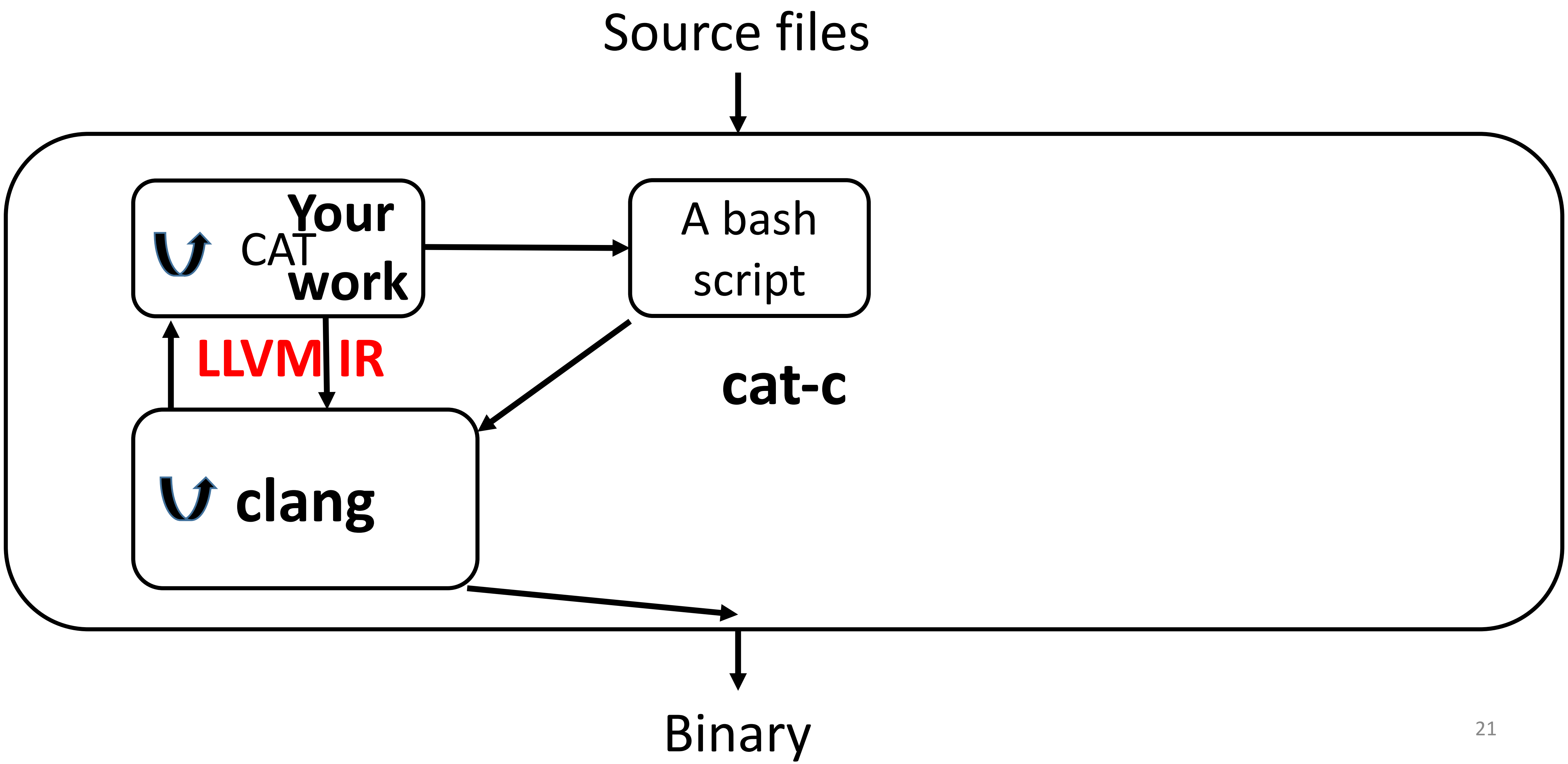
1. Login to a machine
   (e.g., hanlon.wot.eecs.northwestern.edu)
2. Clone the git repository:
   git clone https://github.com/scampanoni/LLVM_middleend_template.git cat-c
3. Compile it and install it:
   cd cat-c ; ./run_me.sh
4. Add the cat-c compiler to your environment
   I. echo "export PATH=~/CAT/bin:$PATH" >> ~/.bash_profile
   II. Logout and login back

Homework: build your own compiler

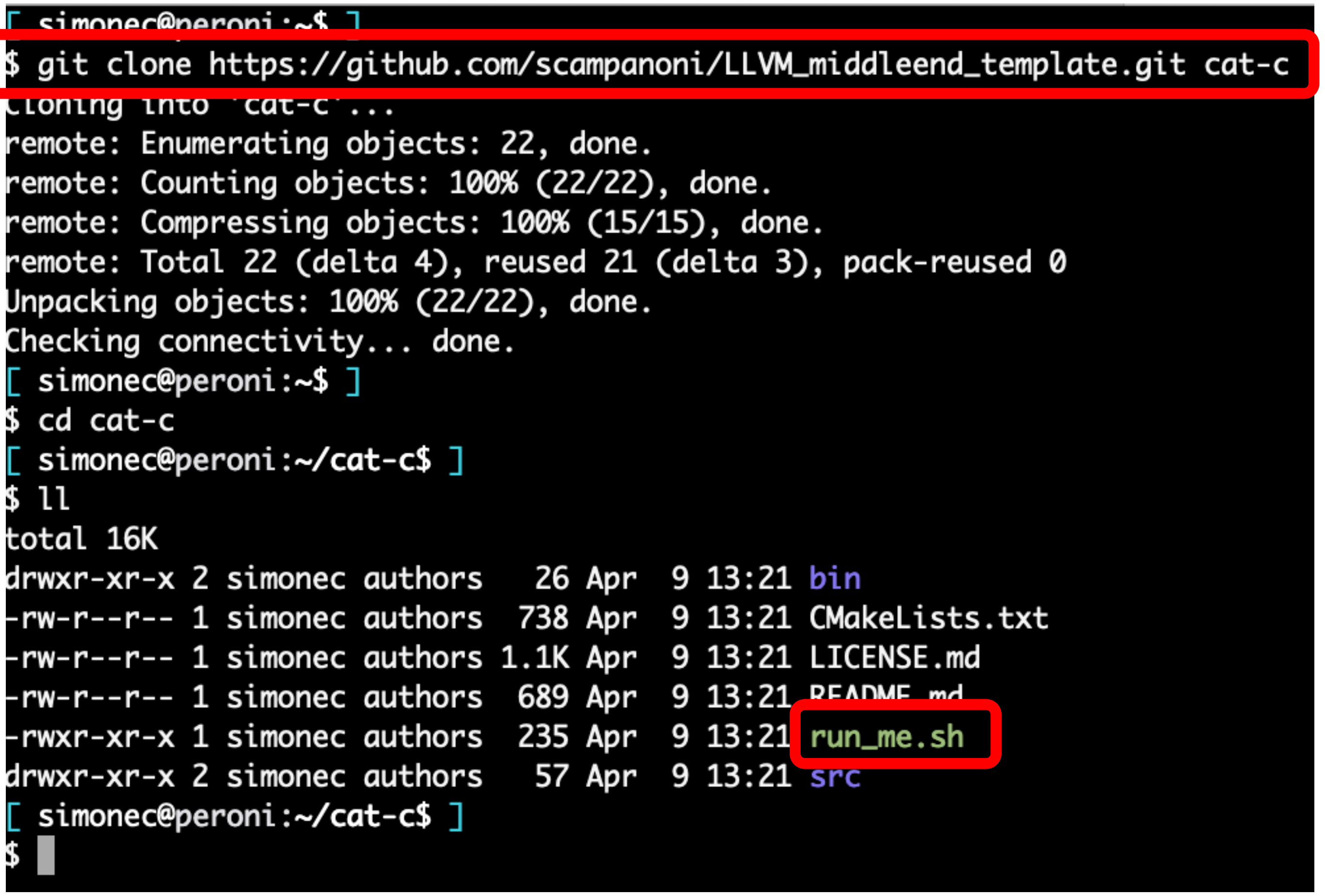
To use cat-c

1. Login to a machine
   (e.g., hanlon.wot.eecs.northwestern.edu)
2. You need to use “cat-c” rather than “clang” in your command line
   (that’s it)
   - For example, if before you run:
     clang myprogram.c –o myprogram
   - Now you need to run:
     cat-c myprogram.c –o myprogram
   - The **only** difference between cat-c and clang is that
     cat-c invokes a new pass at the end of the middle-end

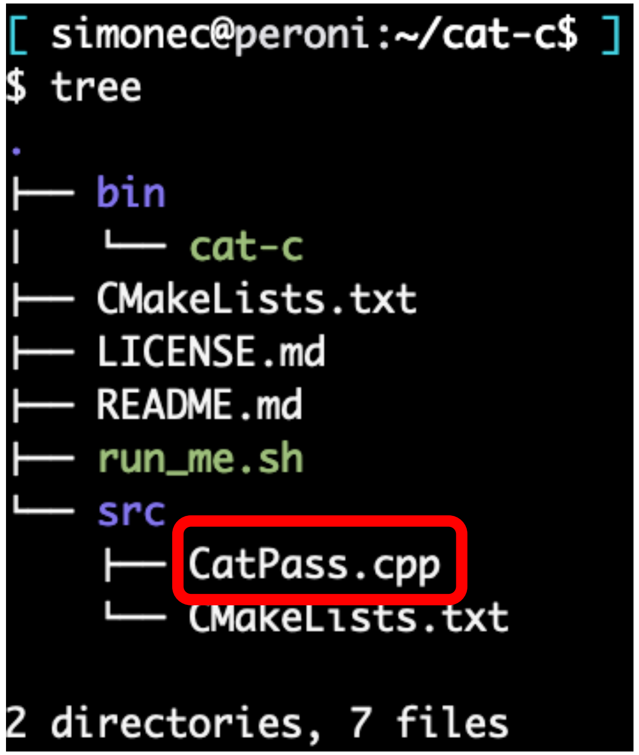
Homework: build your own compiler



The cat-c structure



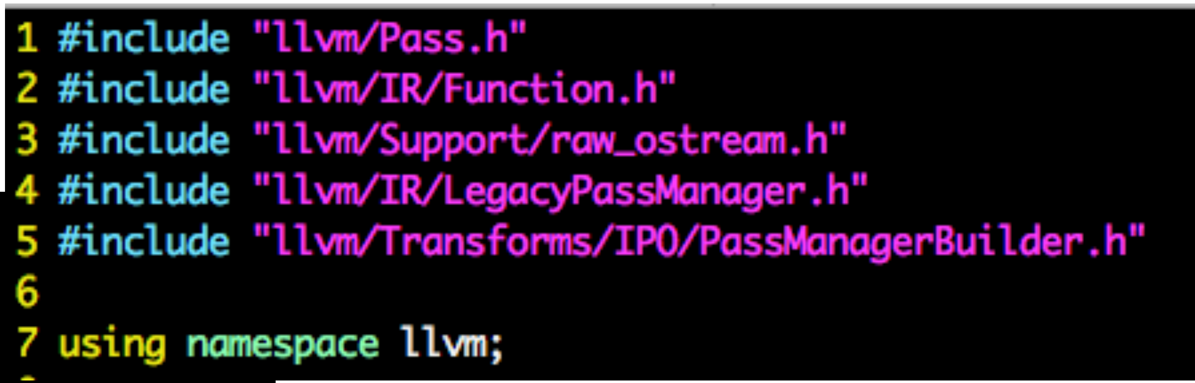
```
[ simonec@peroni:~$ ]
$ git clone https://github.com/scampanoni/LLVM_middleend_template.git cat-c
Cloning into 'cat-c'...
remote: Enumerating objects: 22, done.
remote: Counting objects: 100% (22/22), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 22 (delta 4), reused 21 (delta 3), pack-reused 0
Unpacking objects: 100% (22/22), done.
Checking connectivity... done.
[ simonec@peroni:~$ ]
$ cd cat-c
[ simonec@peroni:~/cat-c$ ]
$ ll
total 16K
drwxr-xr-x 2 simonec authors   26 Apr  9 13:21 bin
-rw-r--r-- 1 simonec authors  738 Apr  9 13:21 CMakeLists.txt
-rw-r--r-- 1 simonec authors 1.1K Apr  9 13:21 LICENSE.md
-rw-r--r-- 1 simonec authors  689 Apr  9 13:21 README.md
-rwxr-xr-x 1 simonec authors  235 Apr  9 13:21 run_me.sh
drwxr-xr-x 2 simonec authors   57 Apr  9 13:21 src
[ simonec@peroni:~/cat-c$ ]
$
```



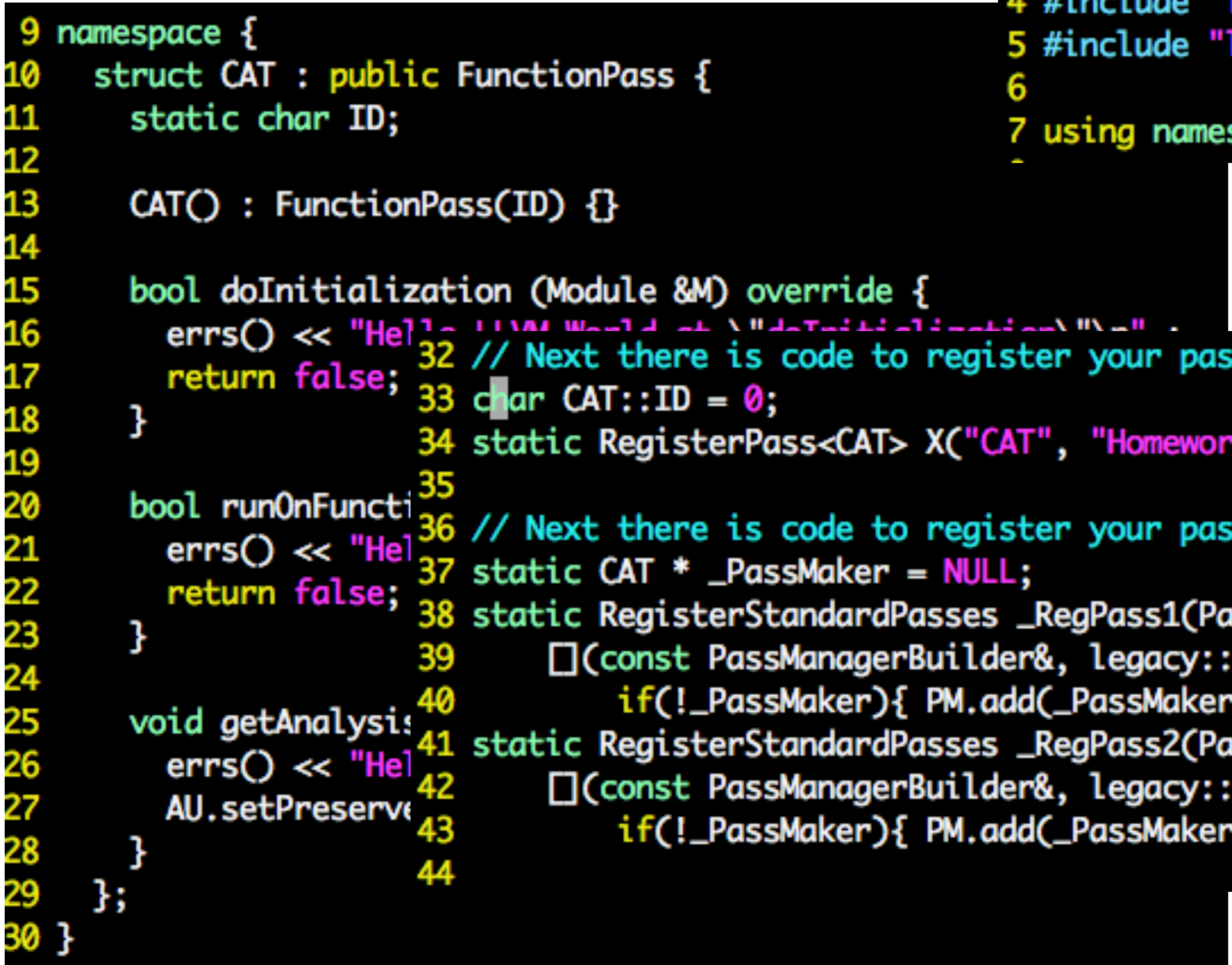
```
[ simonec@peroni:~/cat-c$ ]
$ tree
.
├── bin
│   └── cat-c
├── CMakeLists.txt
├── LICENSE.md
├── README.md
├── run_me.sh
└── src
    ├── CatPass.cpp
    └── CMakeLists.txt

2 directories, 7 files
```

Your CAT work

CatPass.cpp

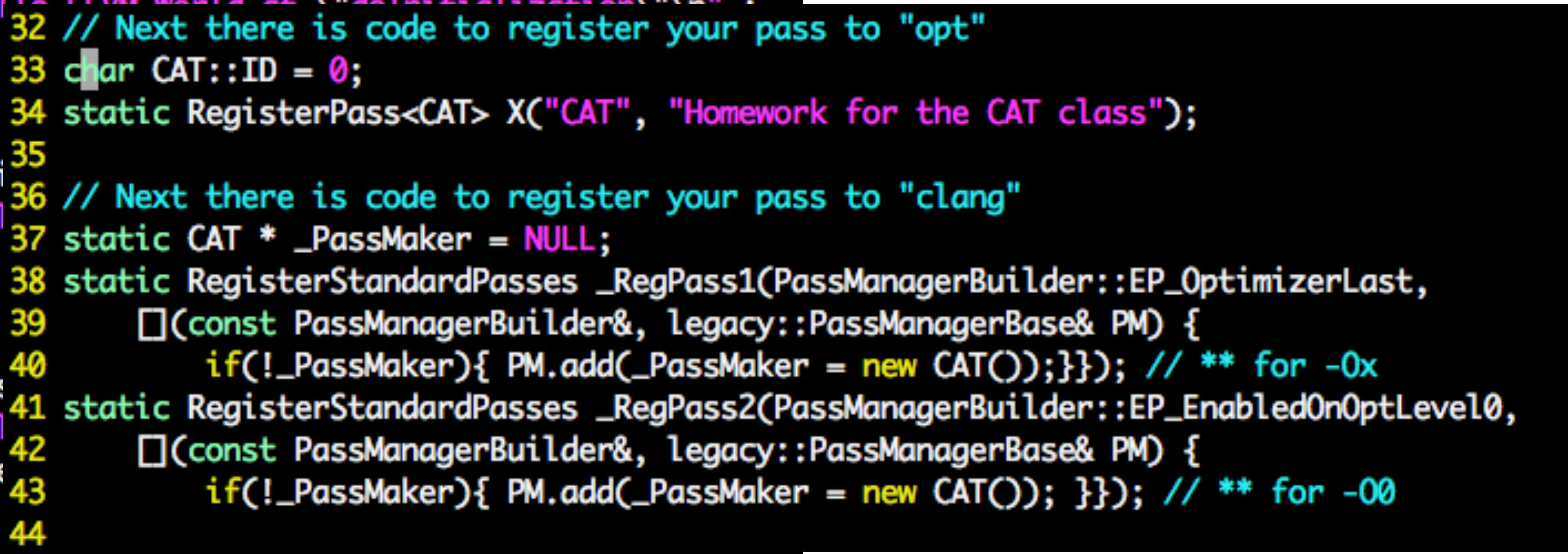


```cpp
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/Transforms/IPO/PassManagerBuilder.h"

using namespace llvm;
```



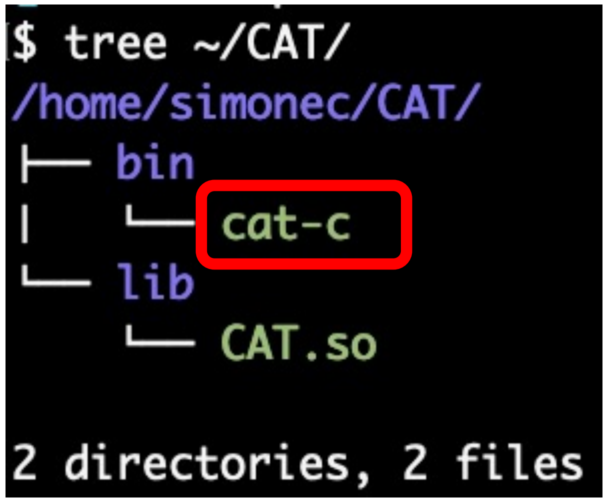
```cpp
namespace {
  struct CAT : public FunctionPass {
    static char ID;

    CAT() : FunctionPass(ID) {}

    bool doInitialization (Module &M) override {
      errs() << "Hello LLVM World at \"doInitialization\"\n" ;
      return false;
    }

    bool runOnFuncti
      errs() << "Hel
      return false;
    }

    void getAnalysis
      errs() << "Hel
      AU.setPreserve
    }
  };
}
```

F.getName()

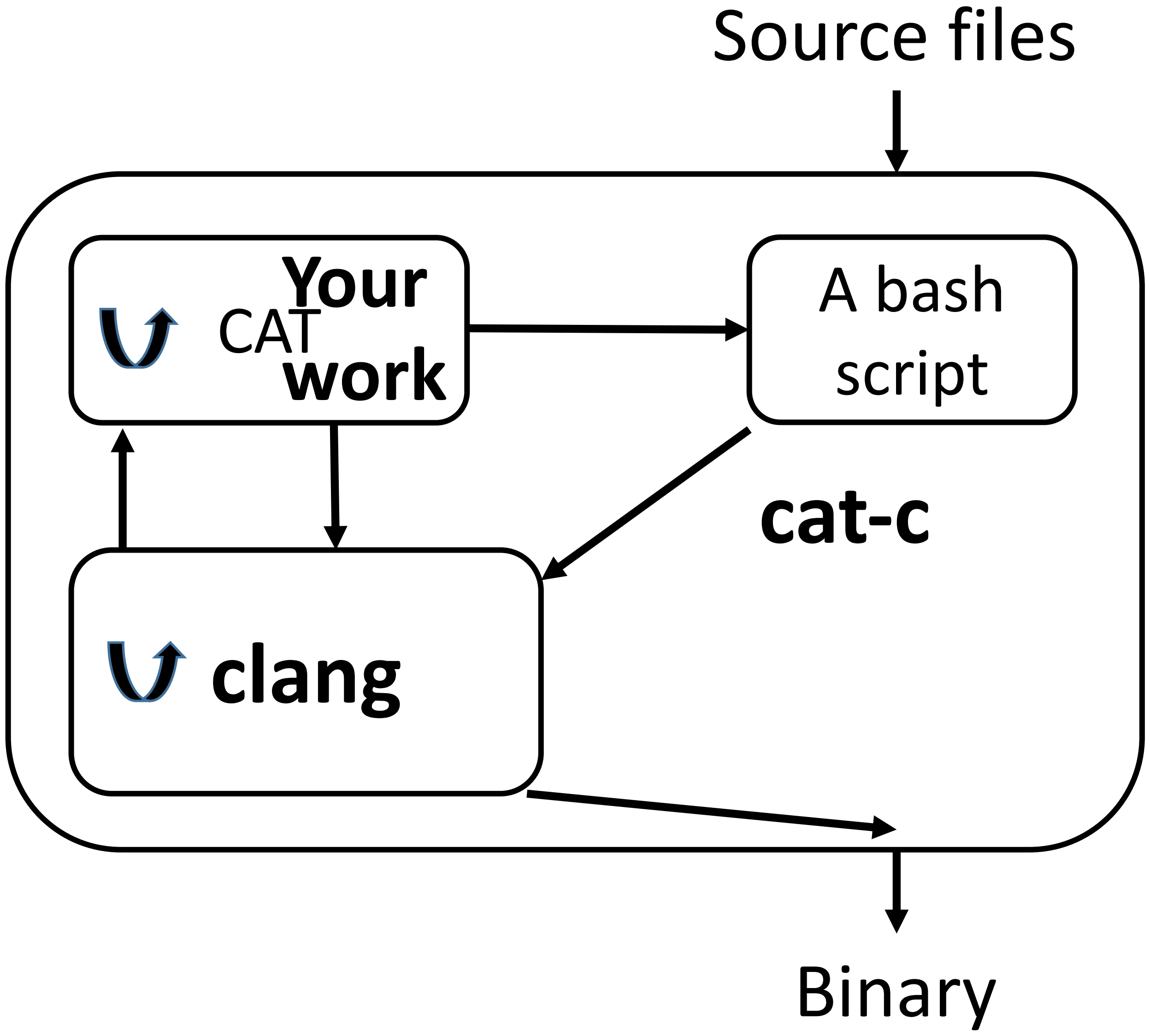


```cpp
// Next there is code to register your pass to "opt"
char CAT::ID = 0;
static RegisterPass<CAT> X("CAT", "Homework for the CAT class");

// Next there is code to register your pass to "clang"
static CAT * _PassMaker = NULL;
static RegisterStandardPasses _RegPass1(PassManagerBuilder::EP_OptimizerLast,
    [](const PassManagerBuilder&, legacy::PassManagerBase& PM) {
        if(!_PassMaker){ PM.add(_PassMaker = new CAT());}}); // ** for -Ox
static RegisterStandardPasses _RegPass2(PassManagerBuilder::EP_EnabledOnOptLevel0,
    [](const PassManagerBuilder&, legacy::PassManagerBase& PM) {
        if(!_PassMaker){ PM.add(_PassMaker = new CAT()); }}); // ** for -O0
```

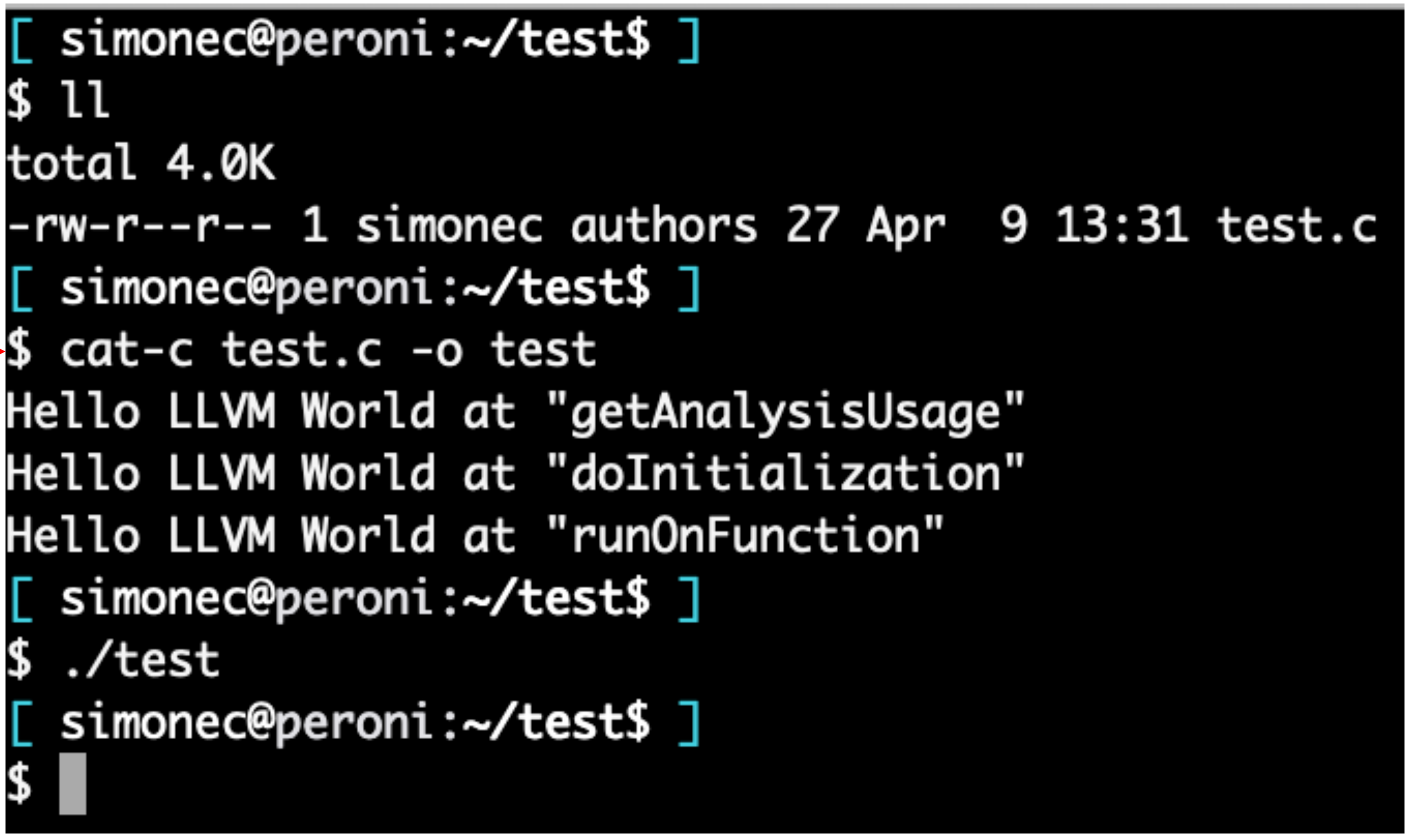
Your cat-c compiler



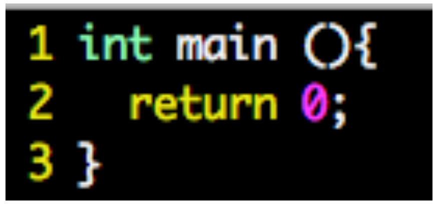
```
$ tree ~/CAT/
/home/simonec/CAT/
├── bin
│   └── cat-c
└── lib
    └── CAT.so

2 directories, 2 files
```



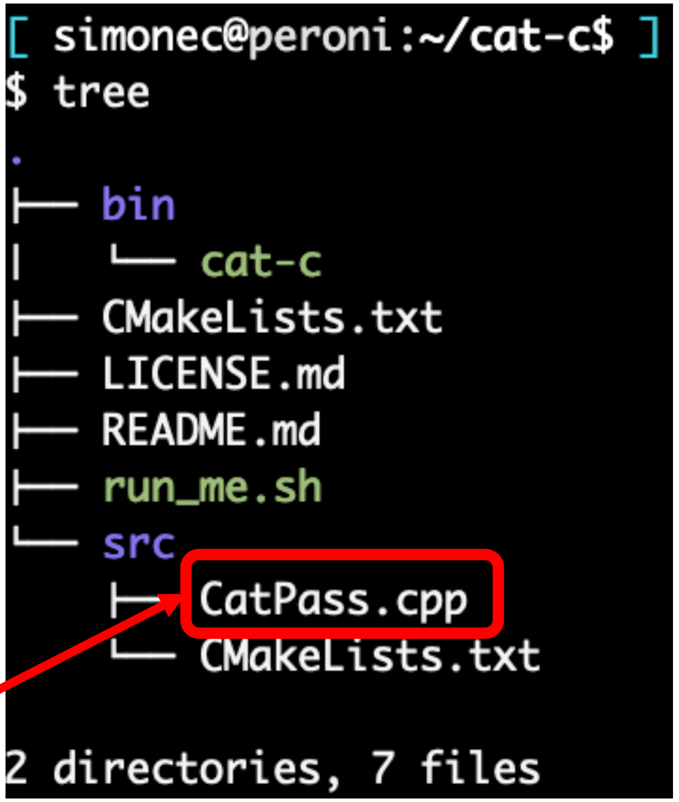
Using your cat-c compiler



```
[ simonec@peroni:~/test$ ]
$ ll
total 4.0K
-rw-r--r-- 1 simonec authors 27 Apr  9 13:31 test.c
[ simonec@peroni:~/test$ ]
$ cat-c test.c -o test
Hello LLVM World at "getAnalysisUsage"
Hello LLVM World at "doInitialization"
Hello LLVM World at "runOnFunction"
[ simonec@peroni:~/test$ ]
$ ./test
[ simonec@peroni:~/test$ ]
$
```



```
int main (){
  return 0;
}
```



```
[ simonec@peroni:~/cat-c$ ]
$ tree
.
├── bin
|   └── cat-c
├── CMakeLists.txt
├── LICENSE.md
├── README.md
├── run_me.sh
└── src
    ├── CatPass.cpp
    └── CMakeLists.txt

2 directories, 7 files
```

To do more than a hello world pass: modify

Homework: build your own compiler

To modify cat-c

1. Modify cat-c/src/CatPass.cpp
   ```
   cd cat-c/build ; vim ../src/CatPass.cpp
   ```

2. Go to the build directory
   ```
   cd cat-c/build
   ```

3. Recompile your CAT and install it
   ```
   make install
   ```

10 assignments: from H0 to H9

- Hi depends on Hi-1
- For every assignment:
    - You have to modify your previous CatPass.cpp
    - You have to pass all tests distributed
- Assignment i: Hi.tar.bz2
    - The description of the homework (Hi.pdf)
    - The tests you have to pass (tests)
- Each assignment is an LLVM pass

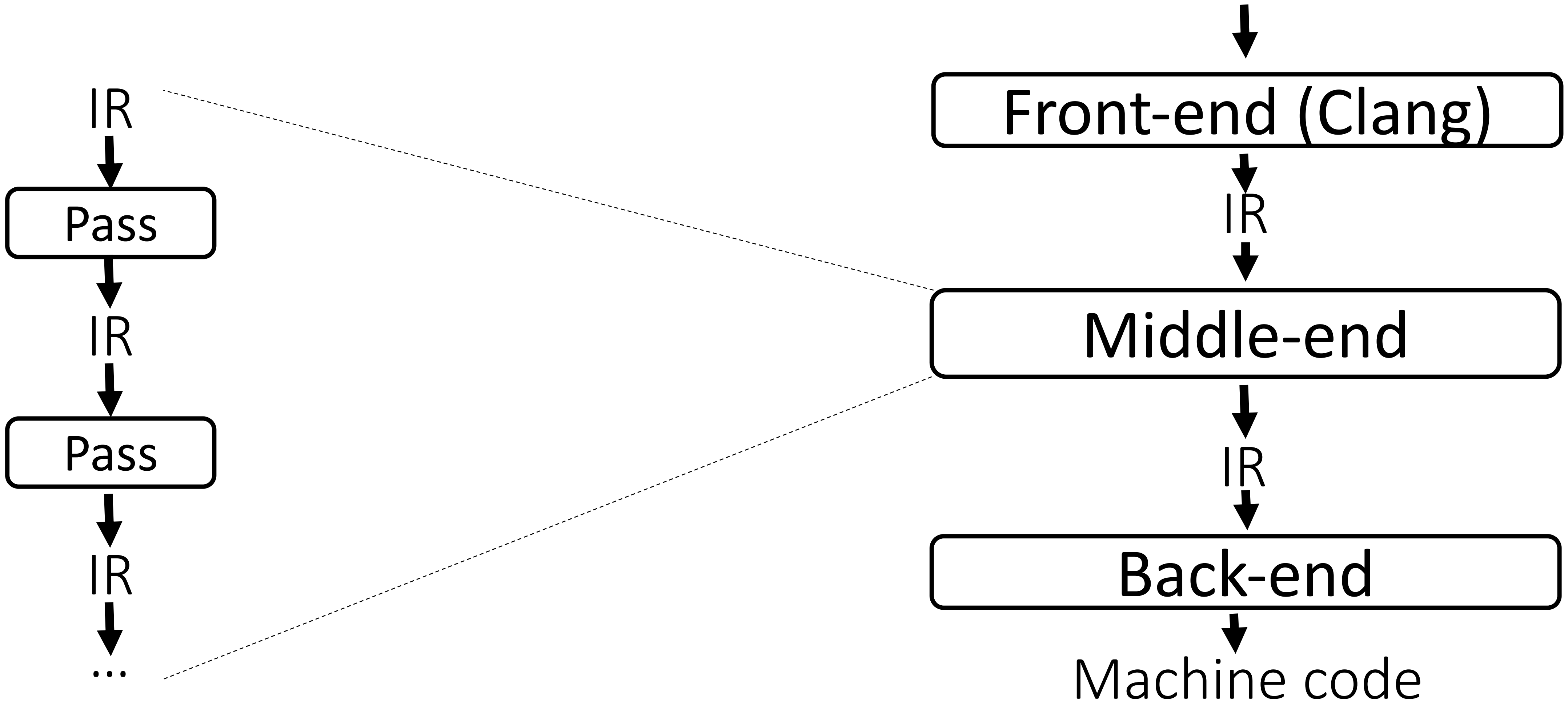
Outline

- Introduction to LLVM

- Homework steps

- Hacking LLVM with CAT

LLVM middle-end is designed around its IR



LLVM tools to read/generate IR

- clang to generate/optimize/translate LLVM IR code
  - To generate binaries from source code or IR code
  - Check Makefile you have in LLVM_introduction.tar.bz2 (Canvas)

- lli to execute (interpret/JIT) LLVM IR code

```
lli FILE.bc
```

- llc to generate assembly from LLVM IR code

```
llc FILE.bc
```

or

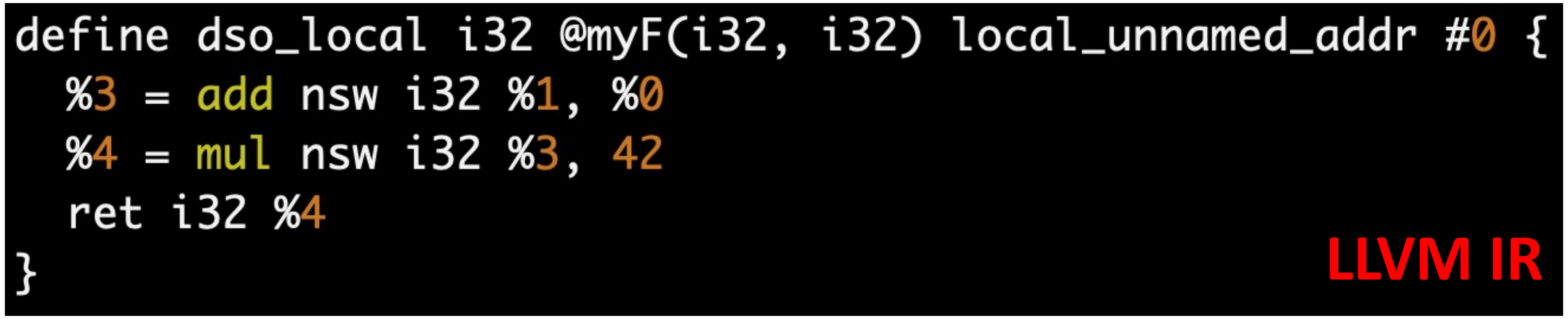
```
clang FILE.bc
```

LLVM tools to read/generate IR

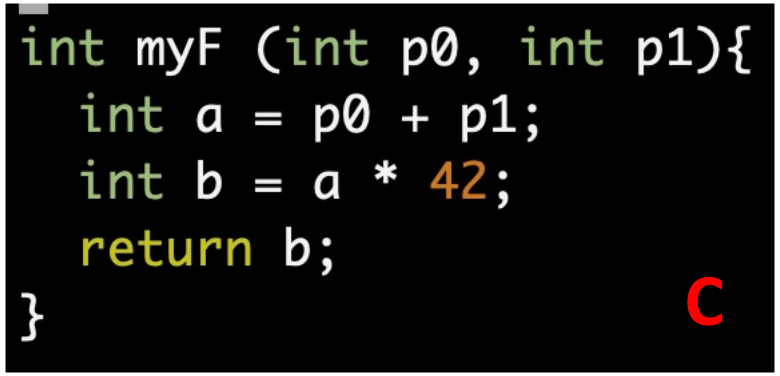
- opt to analyze/transform LLVM IR code
  - Read LLVM IR file
  - Load external passes
  - Run specified passes
  - Respect pass order you specify as input
    - opt -pass1 -pass2 FILE.ll
  - Optionally generate transformed IR
- Useful passes
  - opt -view-cfg FILE.ll
  - opt -view-dom FILE.ll
- opt -help

LLVM IR

- RISC-based
  - Instructions operate on variables



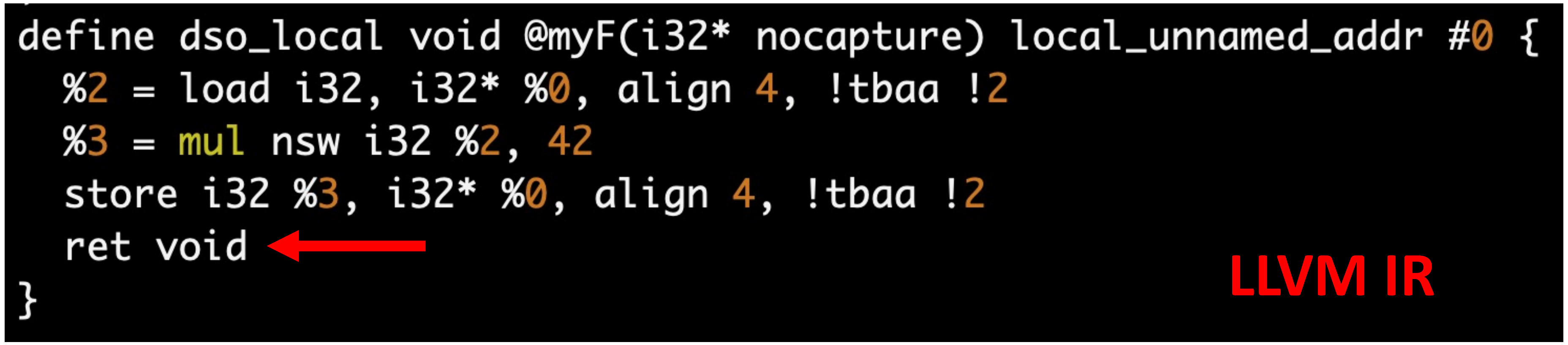
```
define dso_local i32 @myF(i32, i32) local_unnamed_addr #0 {
  %3 = add nsw i32 %1, %0
  %4 = mul nsw i32 %3, 42
  ret i32 %4
}
```



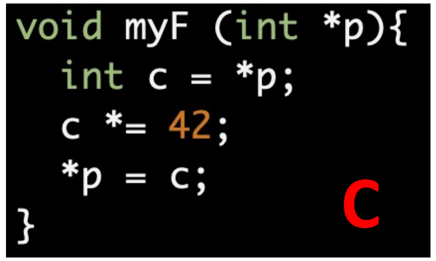
```
int myF (int p0, int p1){
  int a = p0 + p1;
  int b = a * 42;
  return b;
}
```

LLVM IR

- RISC-based
  - Instructions operate on variables
  - Load and store to access memory



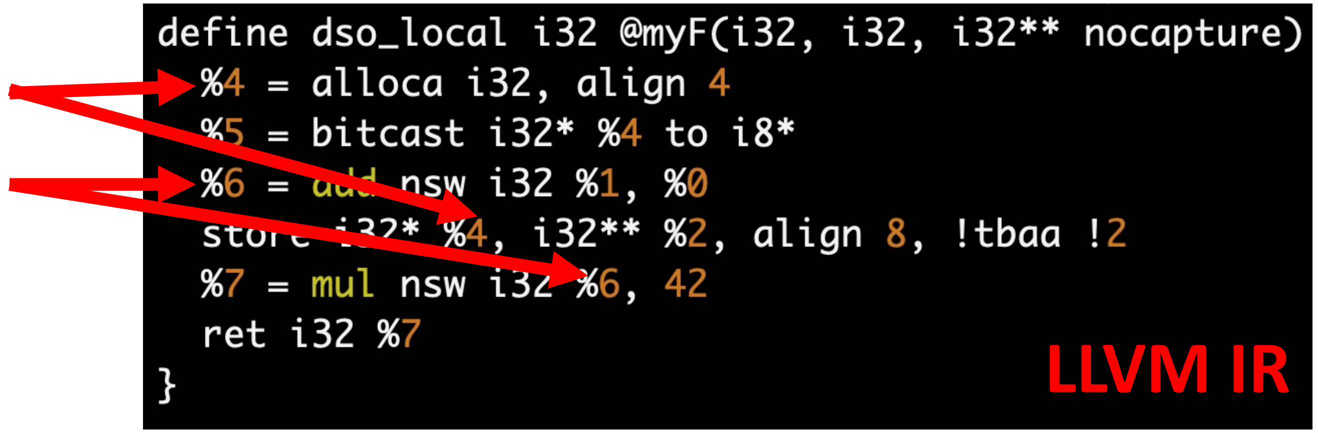
```
define dso_local void @myF(i32* nocapture) local_unnamed_addr #0 {
  %2 = load i32, i32* %0, align 4, !tbaa !2
  %3 = mul nsw i32 %2, 42
  store i32 %3, i32* %0, align 4, !tbaa !2
  ret void
}
```



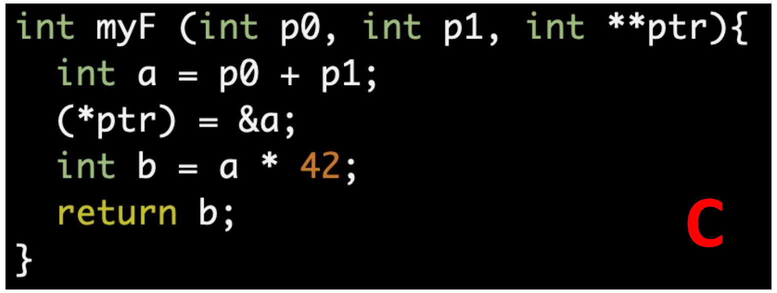
```
void myF (int *p){
  int c = *p;
  c *= 42;
  *p = c;
}
```

LLVM IR

- RISC-based
  - Instructions operate on variables
  - Load and store to access memory



```
define dso_local i32 @myF(i32, i32, i32** nocapture)
  %4 = alloca i32, align 4
  %5 = bitcast i32* %4 to i8*
  %6 = add nsw i32 %1, %0
  store i32* %4, i32** %2, align 8, !tbaa !2
  %7 = mul nsw i32 %6, 42
  ret i32 %7
}
```



```
int myF (int p0, int p1, int **ptr){
  int a = p0 + p1;
  (*ptr) = &a;
  int b = a * 42;
  return b;
}
```

It seems IR variables are 1:1 with C variables but they aren't

LLVM IR

- RISC-based
  - Instructions operate on variables
  - Load and store to access memory

- Include a few high level instructions
  - Function calls (invoke)
  - Pointer arithmetics (getelementptr)
  - Switch semantic (switch)

LLVM IR (2)

- Strongly typed for variables
  - No assignments of variables with different types
  - You need to explicitly cast variables

- No class hierarchy for memory objects

- Variables
  - Global (@myVar)
  - Local to a function (%myVar)
  - Function parameter (define i32 @myF (i32 %myPar))

LLVM IR (3)

- A program is composed by modules (Module), one per source file

```
clang –emit-llvm –c myFile1.c –o myFile1.bc
clang –emit-llvm –c myFile2.c –o myFile2.bc
```

- Modules can be merged

```
llvm-link myFile1.bc myFile2.bc –o mergedModule.bc
```

LLVM IR (4)

LLVM organizes “compiler concepts” in containers

- A module is a container of functions
  - Given an object Module &M

    for (Function &f : M){ }

    Function *sqrtF = M.getFunction(“sqrt”)
  - Given an object Function *f

    Module *m = f->getParent();

- More concepts will come later

LLVM IR (5)

- 3 different (but 100% equivalent) formats
  - Assembly: human-readable format (FILENAME.ll)
  - Bitcode: machine binary on-disk (FILENAME.bc)
  - In memory: in memory binary

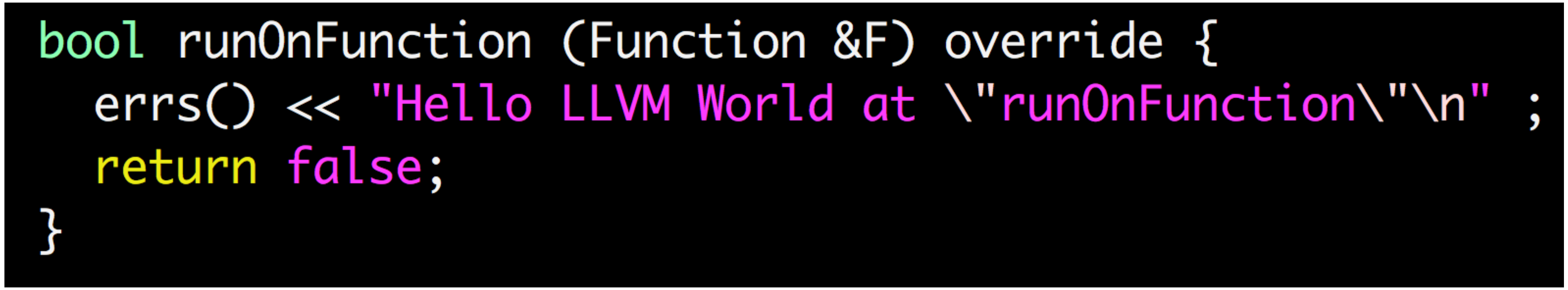
- Generating IR
  - clang for C and C++ languages (similar options w.r.t. GCC)
  - Different front-ends available
    (e.g., flang)

LLVM IR (6)

Print IR concepts: << operator

- To print Function *f
  ```
  errs() << *f << “\n”;
  ```
- To print Function &f
  ```
  errs() << f << “\n”;
  ```
- To print Instruction *i
  ```
  errs() << *i << “\n”;
  ```
- To print Module *m
  ```
  errs() << *m << “\n”;
  ```

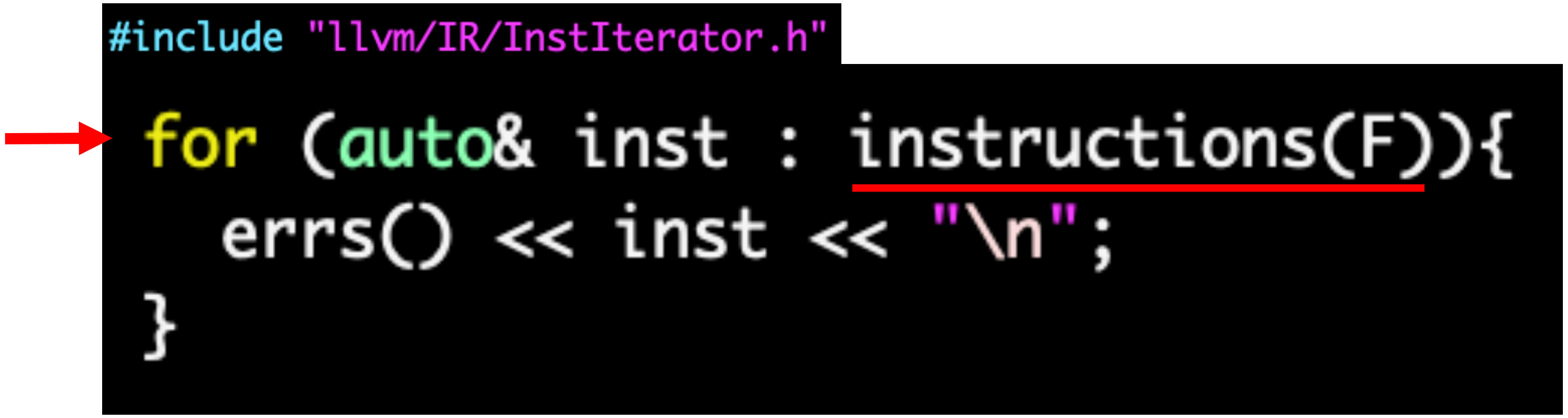
Functions and instructions



```
bool runOnFunction (Function &F) override {
  errs() << "Hello LLVM World at \"runOnFunction\"\n" ;
  return false;
}
```

runOnFunction’s job is to analyze/transform a function F
… by analyzing/transforming its instructions

Functions and instructions



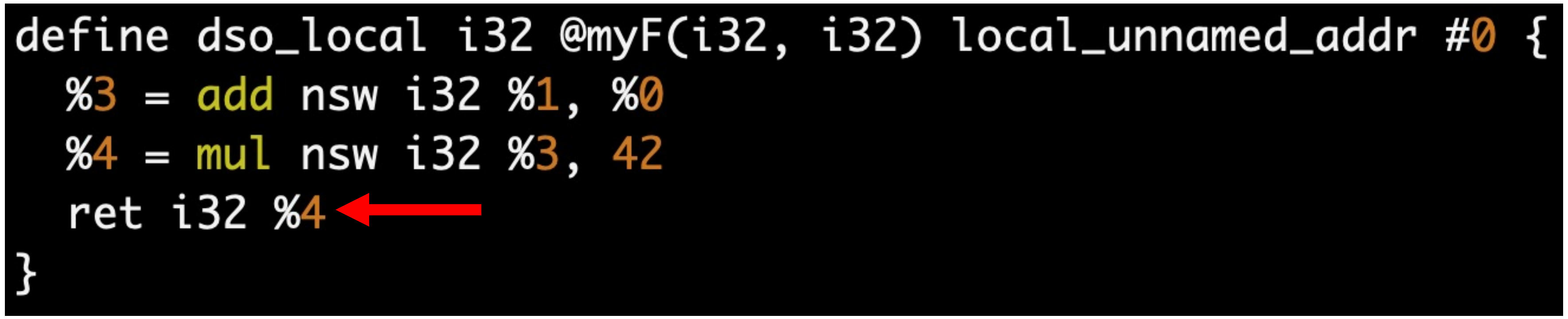
```
#include "llvm/IR/InstIterator.h"
for (auto& inst : instructions(F)){
  errs() << inst << "\n";
}
```

Iteration order:
Follows the order
used to store
instructions
in a function F

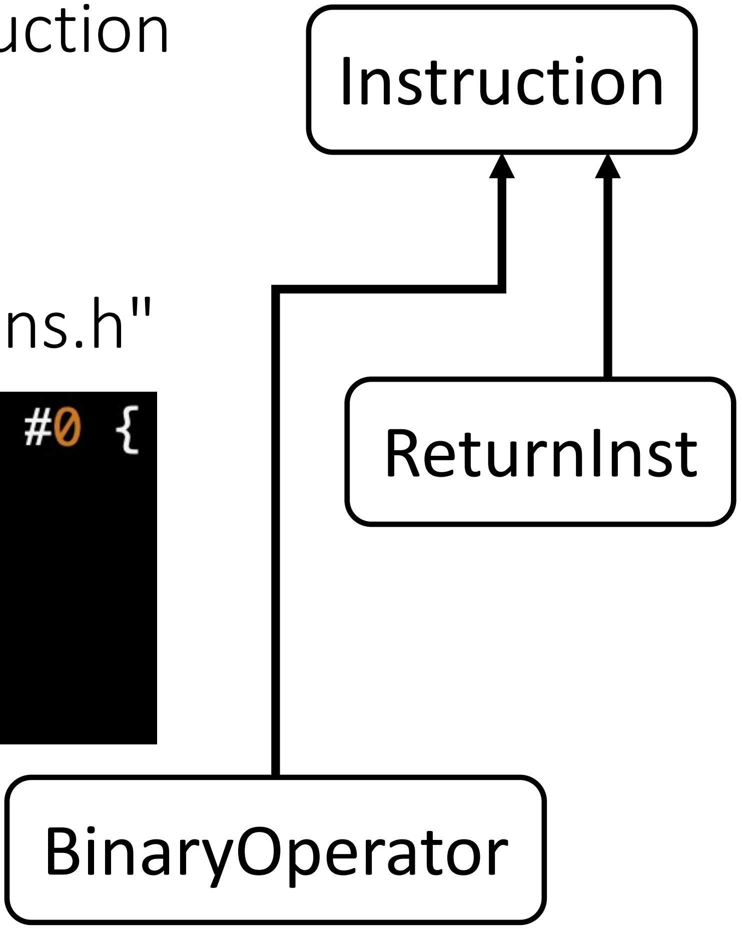
runOnFunction's job is to analyze/transform a function F
… by analyzing/transforming its instructions

Instructions in LLVM

- All instructions are instances of the class llvm::Instruction
- Different instructions are instances of different sub-classes: #include "llvm/IR/Instructions.h"

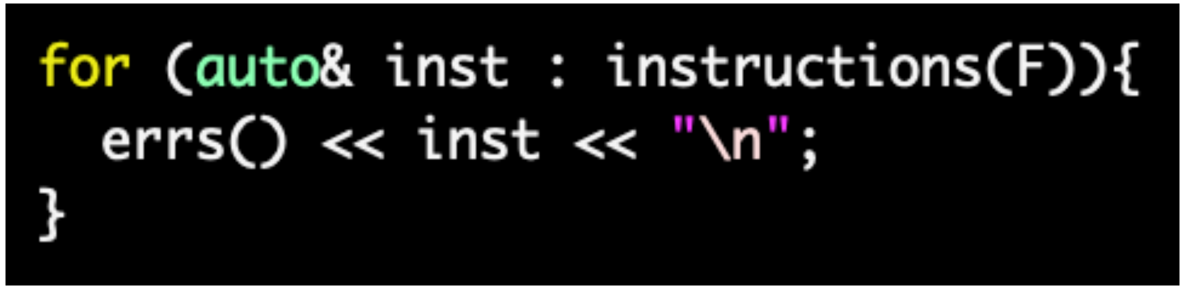


```
define dso_local i32 @myF(i32, i32) local_unnamed_addr #0 {
  %3 = add nsw i32 %1, %0
  %4 = mul nsw i32 %3, 42
  ret i32 %4
}
```

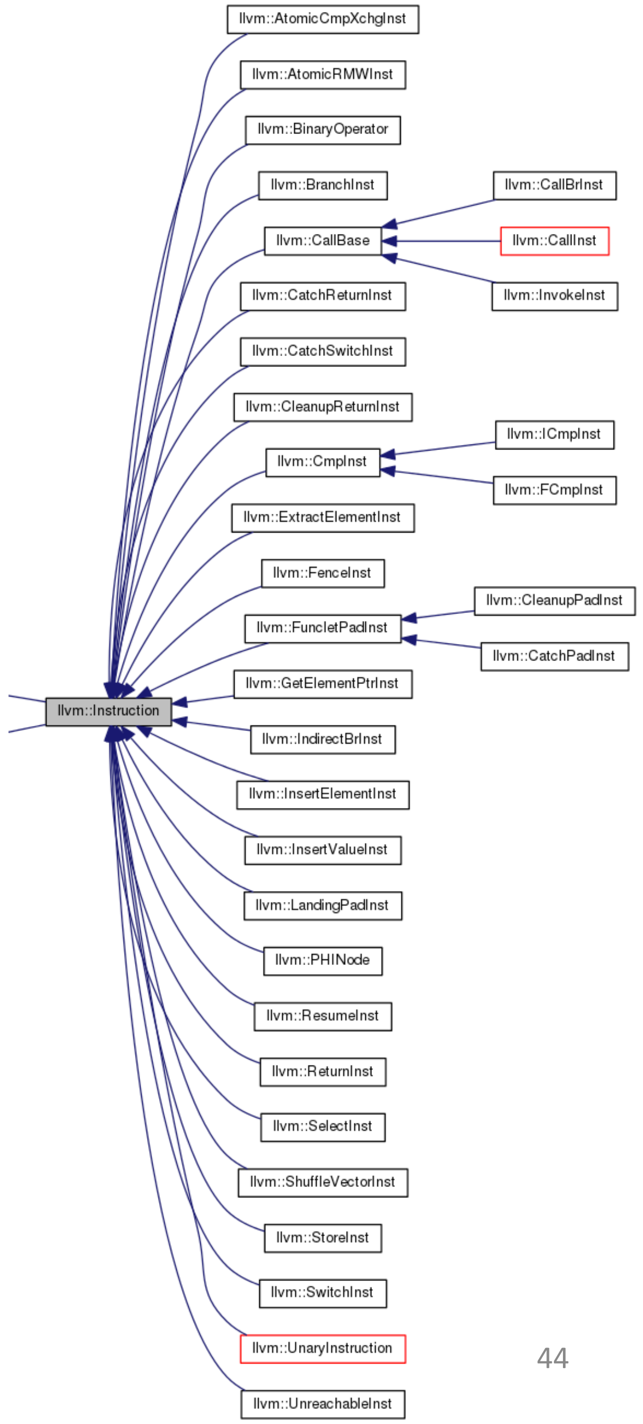


Instructions in LLVM

- All instructions are instances of llvm::Instruction
- Different instructions are instances of different sub-classes
- Each instruction sub-class has extra methods for this type of instructions
  - E.g., Function * CallInst::getCalledFunction()



```
for (auto& inst : instructions(F)){
  errs() << inst << "\n";
}
```



Instructions in LLVM

- You need to cast Instruction objects to access instruction-specific methods
  - **LLVM redefined casting:** #include "llvm/Support/Casting.h"
  - bool isa<CLASS>(objectPointer)

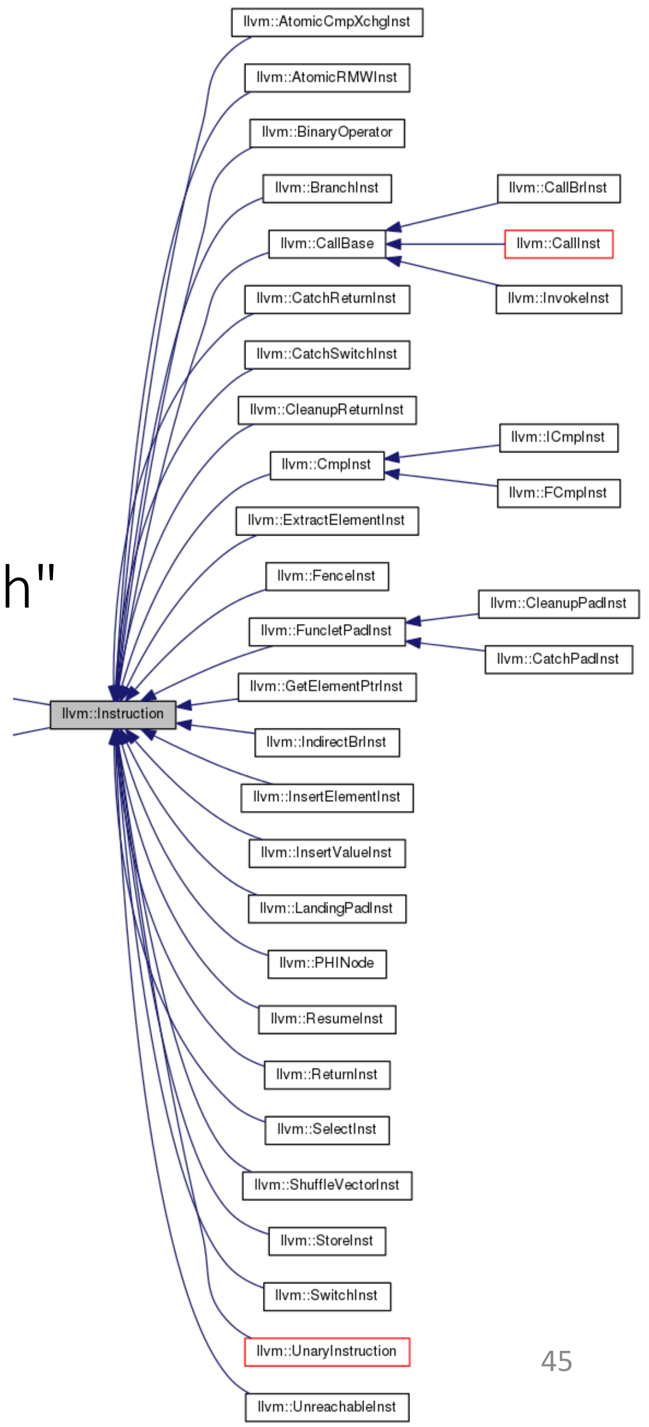
```
for (auto &inst : instructions(&F)){
  if (isa<CallInst>(&inst)){
  }
}
```

  - CLASS *ptrCasted = cast<CLASS>(objectPointer)

```
CallInst *callInst = cast<CallInst>(&I);
Function *callee = callInst->getCalledFunction();
```

  - CLASS *ptrCasted = dyn_cast<CLASS>(objectPointer)

```
for (auto &inst : instructions(&F)){
  CallInst *callInst = dyn_cast<CallInst>(&inst);
  if (callInst != nullptr){
  }
}
```



A great alternative to casting: the visitor pattern

```
#include "llvm/IR/InstVisitor.h"
```

```
class MyInstVisitor : public InstVisitor<MyInstVisitor>{
  public:
    MyInstVisitor(bool enableMyFancyFeature){
      this->enableFeature = enableMyFancyFeature;
    }

    void visitCallInst (CallInst &inst){
      errs() << "CALL = " << inst << "\n";
    }

  private:
    bool enableFeature;
};
```

```
MyInstVisitor wow{true};
wow.visit(F);
```

Now you are ready for your first assignment!

In Canvas: homework/H0.tar.bz2

Test your code in
one of the machine available for this class
(e.g., hanlon.wot.eecs.northwestern.edu)

As Linus Torvalds says …

Talk is cheap. Show me the code.

Let’s start hacking LLVM with CAT

LLVM examples: LLVM_introduction.tar.bz2

Always have faith in your ability

Success will come your way eventually

Best of luck!