LLVM Tutorial

John Criswell
University of Rochester

Overview

History of LLVM

- Developed by Chris Lattner and Vikram Adve at the University of Illinois at Urbana-Champaign
- Released open-source in October 2003
- Default compiler for Mac OS X, iOS, and FreeBSD
- Used by many companies and research groups
- Changed how compiler research is done

Friendly LLVM Community

- LLVM Developer's Mailing List (llvmdev)
  - For LLVM Users and LLVM Developers!
  - I am on this list
  - Helpful if you send from your UR email address
  - Newbies welcome!
- LLVM IRC Channel

LLVM Toolchain Overview

LLVM IR is a language into which programs are translated for analysis and transformation (optimization)

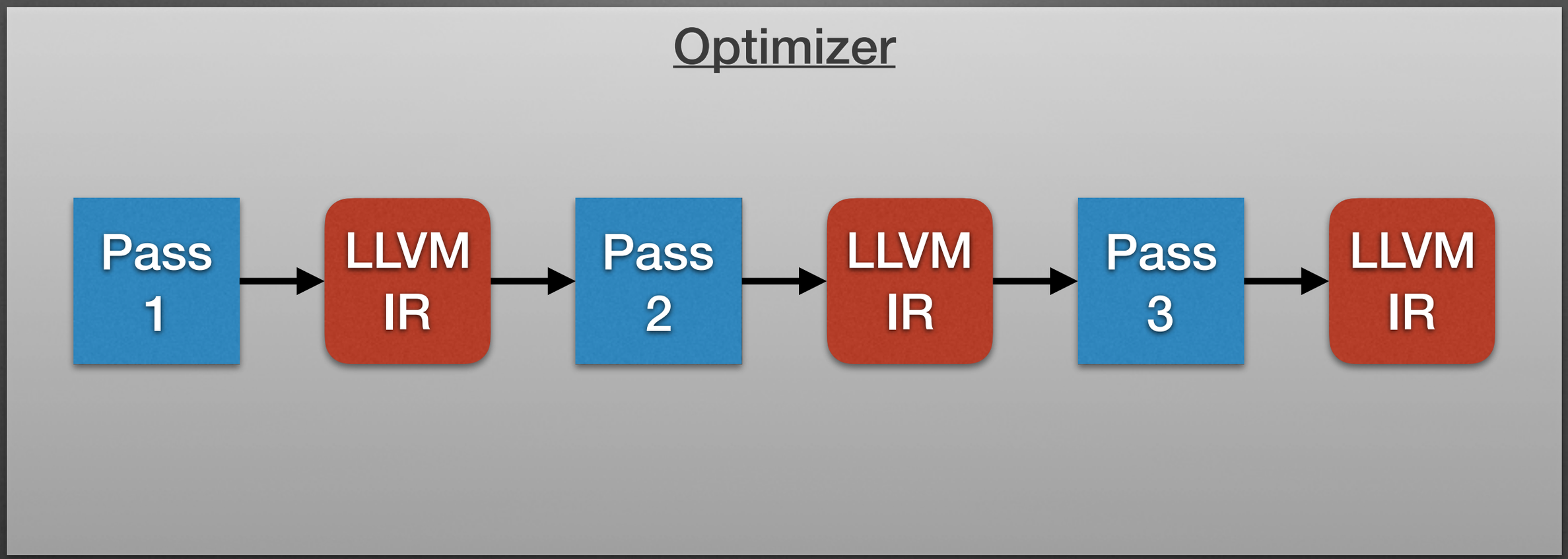
LLVM IR Forms

- LLVM Assembly Language
  - Text form saved on disk for humans to read
- LLVM Bitcode
  - Binary form saved on disk for programs to read
- LLVM In-Memory IR
  - Data structures used for analysis and optimization

LLVM Passes: Separation of Concerns

- Break optimizer into passes
- Each pass performs one analysis or one transform
- LLVM opt tool runs a set of passes on LLVM Bitcode
  - Passes can be loaded as plugins into opt

LLVM Passes



Extending LLVM

Background

- C++
- Know how to use classes, pointers, and references
- Know how to use C++ iterators
- Know how to use Standard Template Library (STL)

Helpful Documents

- LLVM Language Reference Manual
- LLVM Programmer’s Manual
- How to Write an LLVM Pass

The LLVM Language

Overview of LLVM Language

- Each assembly/bitcode file is a Module
- Each Module is comprised of
  - Global variables
  - A set of Functions which are comprised of
    - A set of basic blocks which are comprised of
      - A set of instructions

LLVM Instruction Set

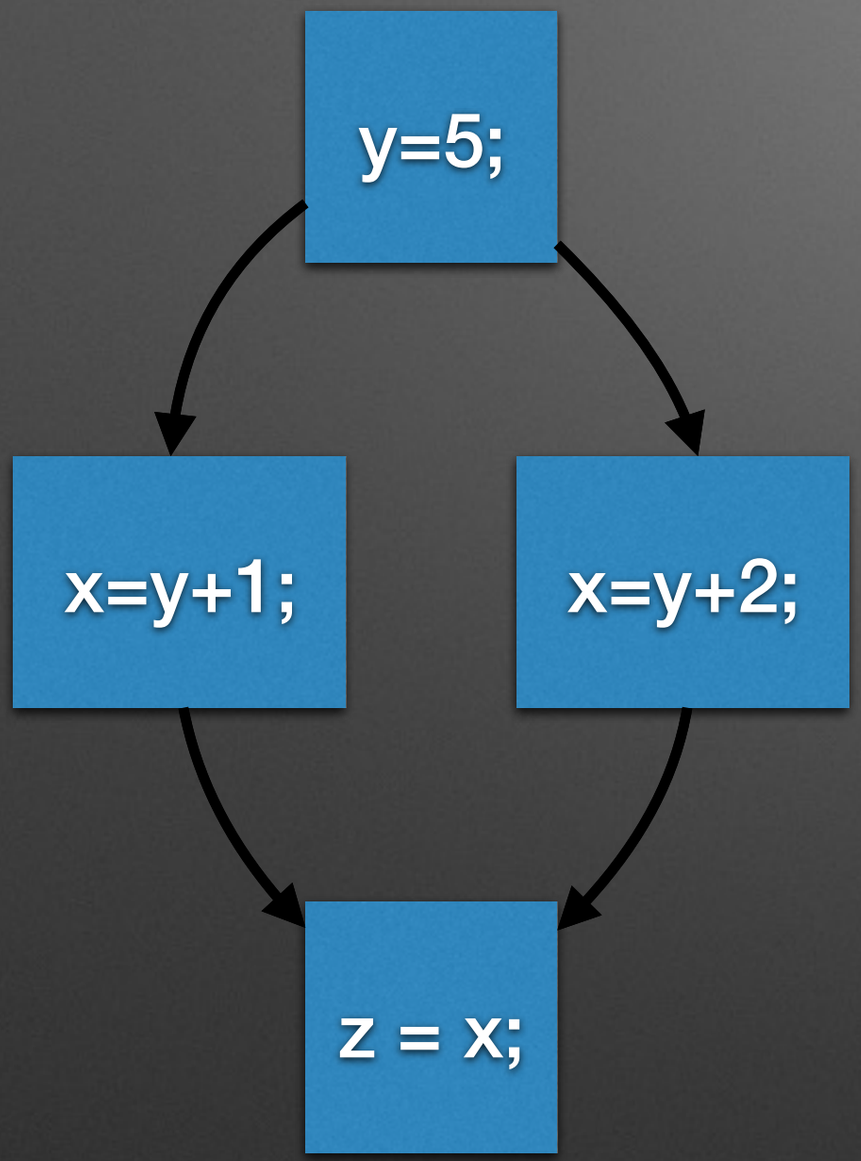
- Arithmetic and binary operators (add, sub, and, or)
- Stack allocation (alloca)
- Memory access instructions (load, store)
- Pointer arithmetic (getelementptr or “GEP”)
- Comparison instructions (icmp, fcmp)
- Terminator instructions for control-flow (br, switch)

Single Static Assignment

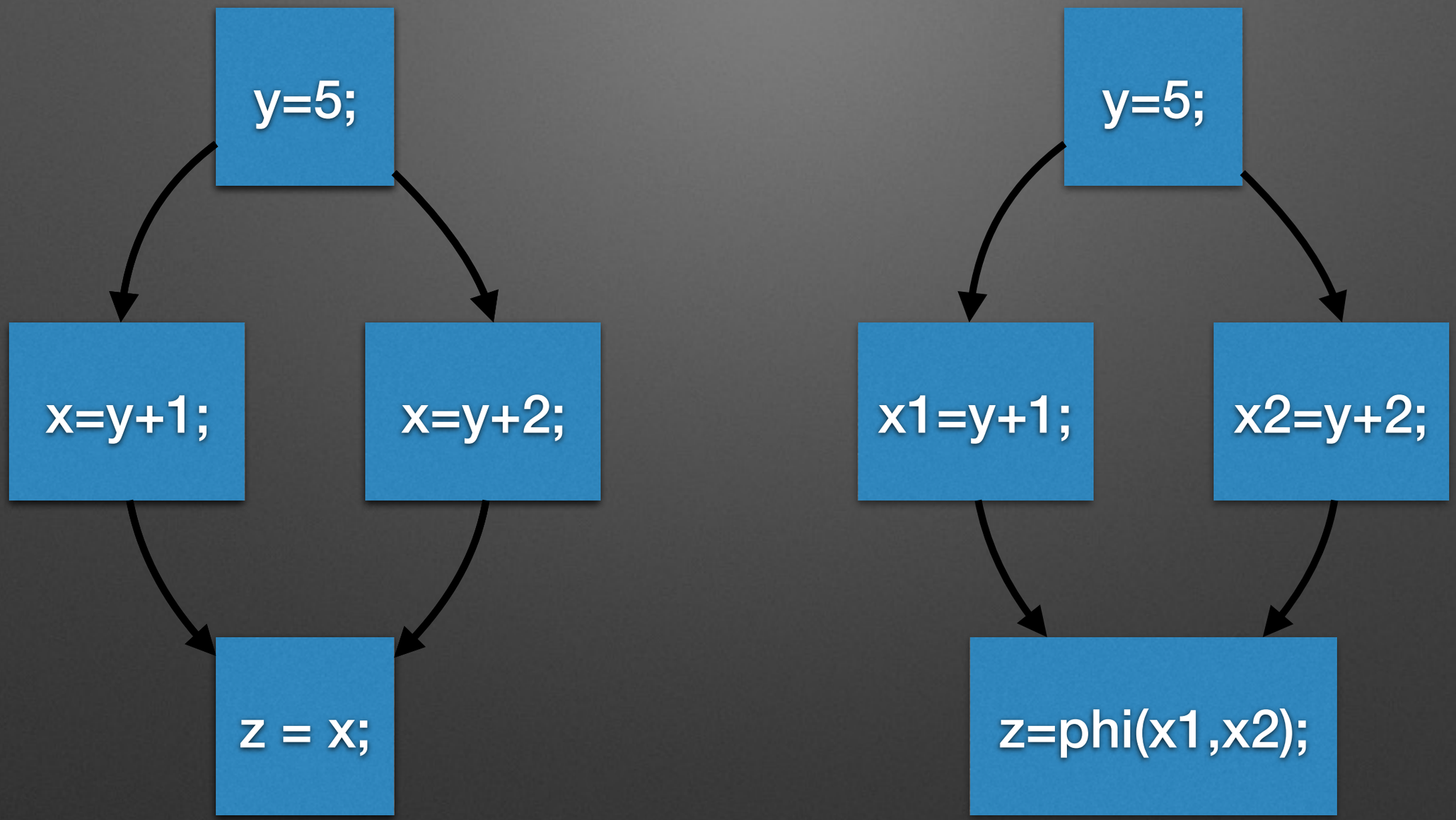
- Each function has infinite set of virtual registers
- Only ***one*** instruction assigns a value to a virtual register (called the ***definition*** of the register)
- An instruction and the register it defines are ***synonymous***

```
%x = add %y, %z
```

The Almighty Phi Node!

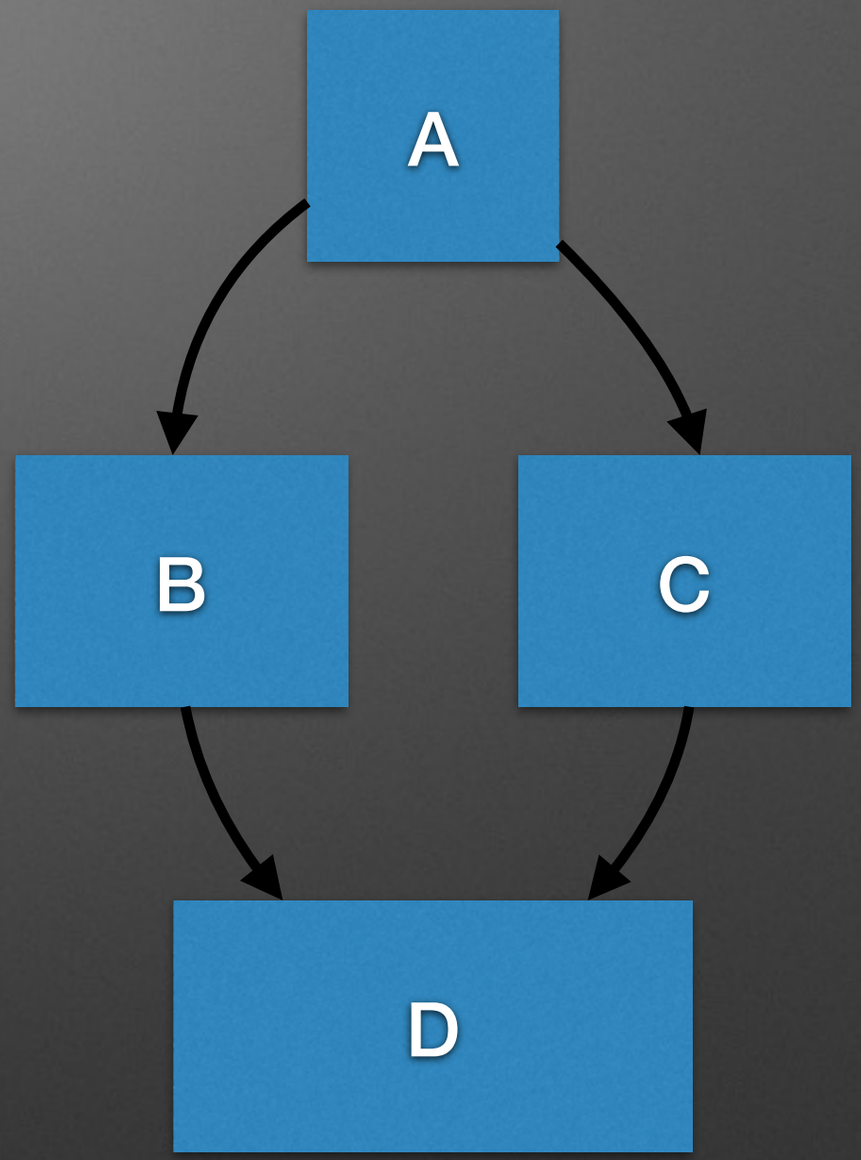


The Almighty Phi Node!



Domination

- The definition of a virtual register must dominate all of its uses (except uses by phi-nodes)
- A dominates B, C, and D



Writing an LLVM Pass

Types of LLVM Passes

- ModulePass
- FunctionPass
- BasicBlockPass

- I recommend ignoring “funny” passes
  - LoopPass
  - RegionPass
- MachineFunctionPass is for manipulating native code

Rules for LLVM Passes

- Only modify data at the scope of the pass
  - ModulePass and examine anything
  - FunctionPass should not modify anything outside of the function
  - BasicBlockPass should not modify anything outside of the basic block

Important Pass Methods: getAnalysisUsage()

- Tells PassManager which analysis passes you need
  - PassManager will schedule analysis passes for you
    - Cannot schedule transform passes this way
- Tells PassManager which analysis results are maintained
  - Avoids re-running expensive analysis passes

runOnModule()

- Entry point for ModulePass
- Passes a reference to the Module
- Can locate functions, basic blocks, globals from Module
- Return true if the pass modifies the program
  - An analysis pass ***always*** returns false.
  - A transform pass can return either true or false.

runOnFunction()

- Called for each function in the Module
- Passed reference to Function
- Return false for no modifications; true for modifications

runOnBasicBlock()

- You get the idea…

LLVM Class Hierarchy

- Available via Doxygen
- Hierarchal Inheritance Tree
  - llvm::Value -> llvm::Instruction -> llvm::LoadInst
- Every instruction has its own class
- Nearly everything that is a virtual register is a subclass of llvm::Value

Pass Strategy: Separation of Concerns

- Don’t add code in SSA form
  - Add local variables via alloca
  - Add loads/stores to read/write variable
  - mem2reg pass converts alloca into virtual register
- Don’t do constant propagation
- Don’t do dead code elimination

Seeking Help

Helpful Documents

- LLVM Language Reference Manual
- LLVM Programmer’s Manual
- How to Write an LLVM Pass
- Doxygen (documents classes and methods)