Lecture 3
Overview of the LLVM Compiler

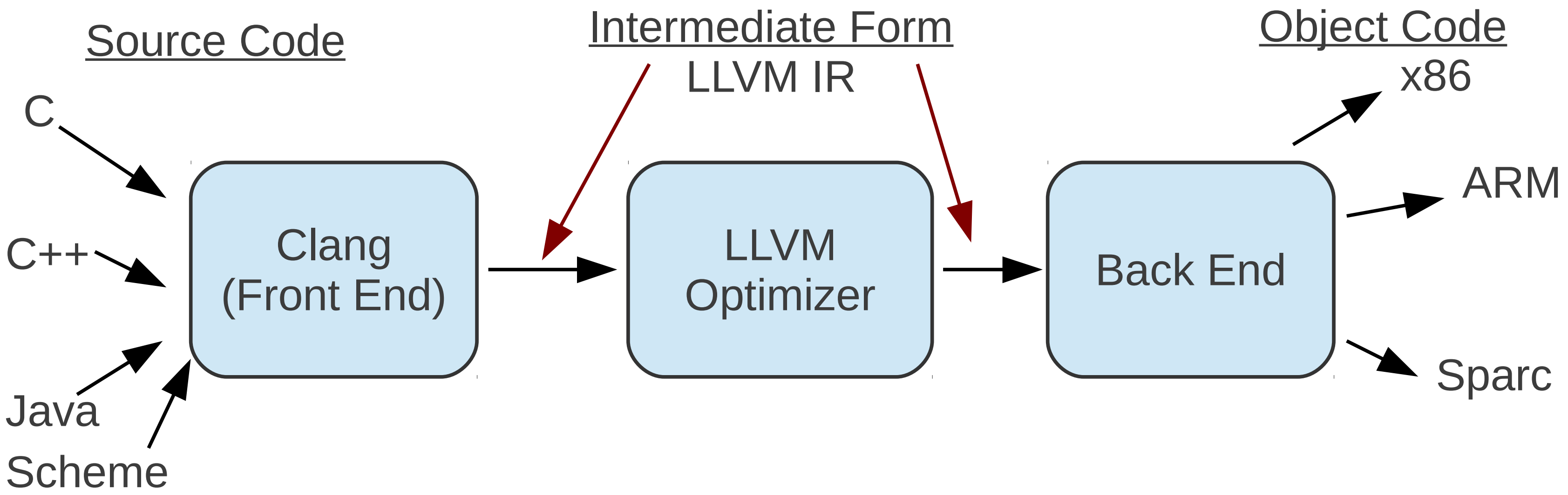
Deby Katz
Substantial portions courtesy of Gennady Pekhimenko, Olatunji Ruwase, Chris Lattner, Vikram Adve, and David Koes

LLVM Compiler System

- **The LLVM Compiler Infrastructure**
  - Provides reusable components for building compilers
  - Reduce the time/cost to build a new compiler
  - Build different kinds of compilers
    - Our homework assignments focus on static compilers
    - There are also JITs, trace-based optimizers, etc.

- **The LLVM Compiler Framework**
  - End-to-end compilers using the LLVM infrastructure
  - Support for C and C++ is robust and aggressive:
  - Java, Scheme and others are in development
  - Emit C code or native code for X86, Sparc, PowerPC

Visualizing the LLVM Compiler System



- **The LLVM Optimizer is a series of "passes"**
  - Analysis and optimization passes, run one after another
  - Analysis passes do not change code, optimization passes do
- **LLVM Intermediate Form is a *Virtual Instruction Set***
  - Language- and target-independent form
    - Used to perform the same passes for all source and target languages
  - Internal Representation (IR) and external (persistent) representation

Tutorial Overview

- **Introduction to the running example**
- **LLVM C/C++ Compiler Overview**
  - High-level view of an example LLVM compiler
- **The LLVM Virtual Instruction Set**
  - IR overview and type-system
- **Coding in LLVM**
- **Important LLVM Tools**
  - opt, code generator, JIT, test suite, bugpoint

Running Example: Constant Folding

```
int subFiveThree() {
  int a = 5;
  int b = 3;
  return a - b;
}
```

Literal parsing to LLVM IR

```
; Function Attrs: nounwind
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 5, i32* %a, align 4
  store i32 3, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %sub = sub nsw i32 %0, %1
  ret i32 %sub
}
```

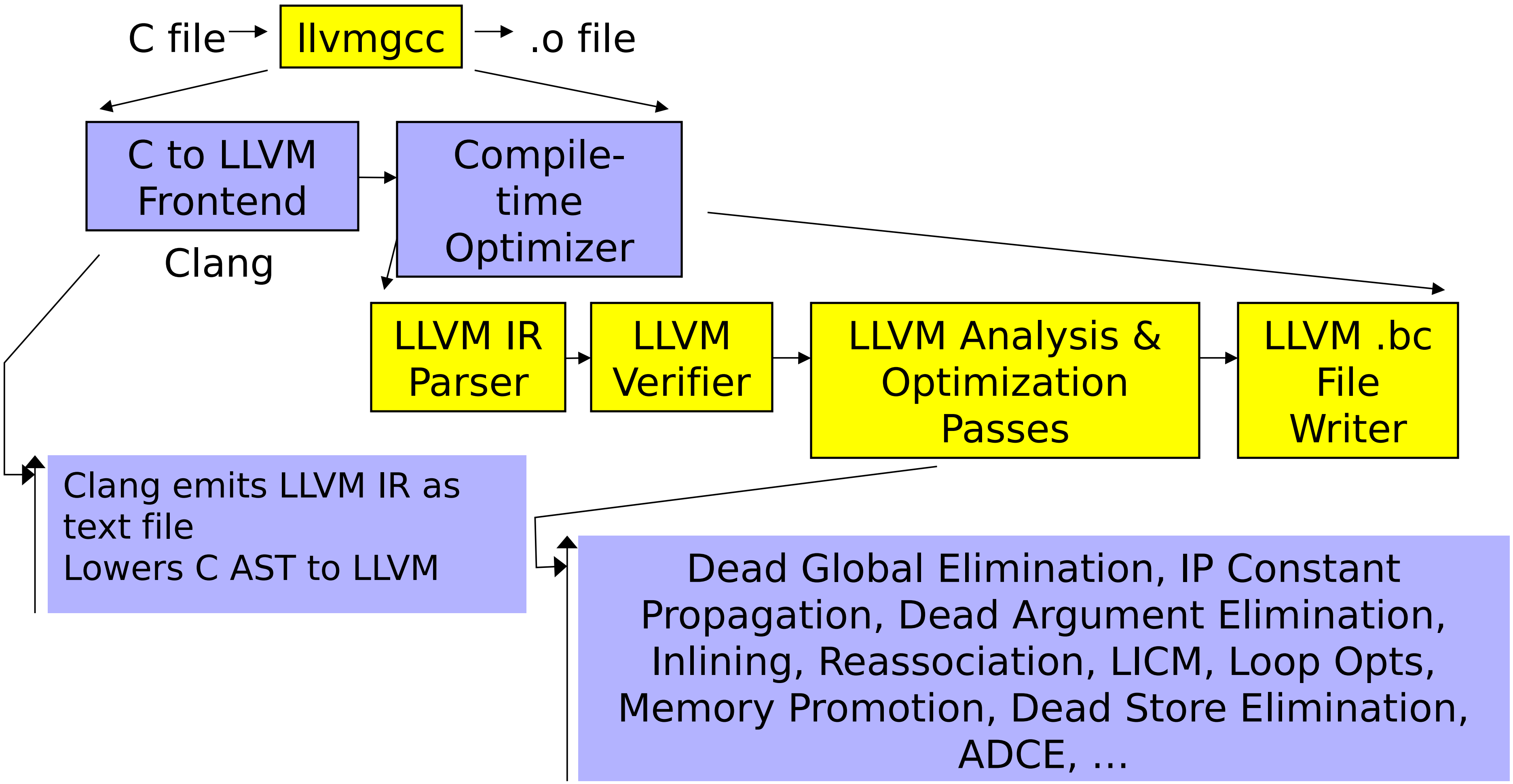
We want to optimize to:

```
int subFiveThree()
{
  return 2;
}
```

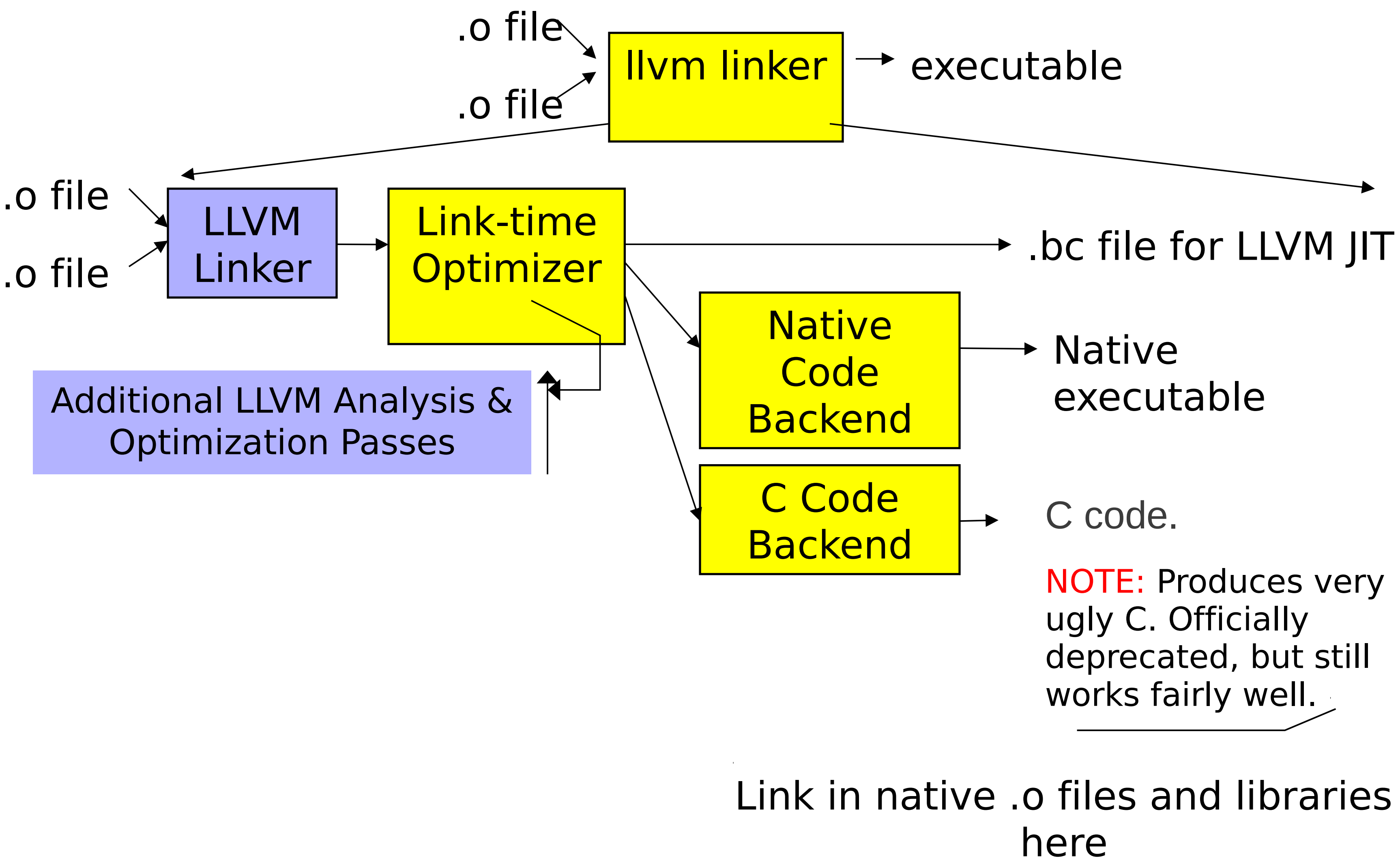
Tutorial Overview

- **Introduction to the running example**
- **LLVM C/C++ Compiler Overview**
  - High-level view of an example LLVM compiler
- **The LLVM Virtual Instruction Set**
  - IR overview and type-system
- **Coding in LLVM**
- **Important LLVM Tools**
  - opt, code generator, JIT, test suite, bugpoint

Looking into events at compile-time



Looking into events at link-time



Tutorial Overview

- **Introduction to the running example**
- **LLVM C/C++ Compiler Overview**
  - High-level view of an example LLVM compiler
- **The LLVM Virtual Instruction Set**
  - IR overview and type-system
- **Coding in LLVM**
- **Important LLVM Tools**
  - opt, code generator, JIT, test suite, bugpoint

Goals of LLVM Intermediate Representation (IR)

- **Easy to produce, understand, and define**
- **Language- and Target-Independent**
- **One IR for analysis and optimization**
  - Supports high- ***and*** low-level optimization
- **Optimize as much as early as possible**

LLVM Instruction Set Overview

- **Low-level and target-independent semantics**
  - RISC-like three address code
  - Infinite virtual register set in SSA form
  - Simple, low-level control flow constructs
  - Load/store instructions with typed-pointers
- **IR has text, binary, and in-memory forms**

```
for (i = 0; i < N; i++)
   Sum(&A[i], &P);
```

```
loop:                ; preds = %bb0, %loop
  %i.1 = phi i32 [ 0, %bb0 ], [ %i.2, %loop ]
  %AiAddr = getelementptr float* %A, i32 %i.1
  call void @Sum(float %AiAddr, %pair* %P)
  %i.2 = add i32 %i.1, 1
  %exitcond = icmp eq i32 %i.1, %N
  br i1 %exitcond, label %outloop, label %loop
```

LLVM Instruction Set Overview (Continued)

- **High-level information exposed in the code**
  - Explicit dataflow through SSA form
  - (more on SSA later in the course)
  - Explicit control-flow graph (even for exceptions)
  - Explicit language-independent type-information
  - Explicit typed pointer arithmetic
  - Preserve array subscript and structure indexing

```
for (i = 0; i < N; i++)
   Sum(&A[i], &P);
```

```
loop:                    ; preds = %bb0, %loop
  %i.1 = phi i32 [ 0, %bb0 ], [ %i.2, %loop ]
  %AiAddr = getelementptr float* %A, i32 %i.1
  call void @Sum(float %AiAddr, %pair* %P)
  %i.2 = add i32 %i.1, 1
  %exitcond = icmp eq i32 %i.1, %N
  br i1 %exitcond, label %outloop, label %loop
```

Lowering source-level types to LLVM

- **Source language types are lowered:**
  - Rich type systems expanded to simple types
  - Implicit & abstract types are made explicit & concrete
- **Examples of lowering:**
  - References turn into pointers: **`T&`** → **`T*`**
  - Complex numbers: **`complex float`** → **`{ float, float }`**
  - Bitfields: **`struct X { int Y:4; int Z:2; }`** → **`{ i32 }`**
- The entire type system consists of:
  - Primitives: label, void, float, integer, ...
    - Arbitrary bitwidth integers (i1, i32, i64)
  - Derived: pointer, array, structure, function
  - No high-level types: type-system is language neutral
- Type system allows arbitrary casts:
  - Allows expressing weakly-typed languages, like C

LLVM Program Structure

- **Module contains Functions and GlobalVariables**
  - Module is unit of compilation, analysis, and optimization
- **Function contains BasicBlocks and Arguments**
  - Functions roughly correspond to functions in C
- **BasicBlock contains list of instructions**
  - Each block ends in a control flow instruction
- **Instruction is opcode + vector of operands**

Our Example, Compiled to LLVM

```
int subFiveThree() {
  int a = 5;
  int b = 3;
  return a - b;
}
```

```
; Function Attrs: nounwind
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 5, i32* %a, align 4
  store i32 3, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %sub = sub nsw i32 %0, %1
  ret i32 %sub
}
```

Our Example, Compiled to LLVM

```
int subFiveThree() {
  int a = 5;
  int b = 3;
  return a - b;
}
```

```
; Function Attrs: nounwind
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 5, i32* %a, align 4
  store i32 3, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %sub = sub nsw i32 %0, %1
  ret i32 %sub
}
```

- Stack allocation is explicit in LLVM

Our Example, Compiled to LLVM

```
int subFiveThree() {
  int a = 5;
  int b = 3;
  return a - b;
}
```

```
; Function Attrs: nounwind
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 5, i32* %a, align 4
  store i32 3, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %sub = sub nsw i32 %0, %1
  ret i32 %sub
}
```

- All loads and stores are explicit in the LLVM representation

Our Example, Compiled to LLVM

```
int subFiveThree() {
  int a = 5;
  int b = 3;
  return a - b;
}
```

```
; Function Attrs: nounwind
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 5, i32* %a, align 4
  store i32 3, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %sub = sub nsw i32 %0, %1
  ret i32 %sub
}
```

- Instruction results can be referred to as values

Our Example: Desired Transformation

```
; Function Attrs: nounwind
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 5, i32* %a, align 4
  store i32 3, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %sub = sub nsw i32 %0, %1
  ret i32 %sub
}
```

```
entry:
  ret i32 2
}
```

Tutorial Overview

- **Introduction to the running example**
- **LLVM C/C++ Compiler Overview**
  - High-level view of an example LLVM compiler
- **The LLVM Virtual Instruction Set**
  - IR overview and type-system
- **Coding in LLVM**
- **Important LLVM Tools**
  - opt, code generator, JIT, test suite, bugpoint

LLVM Coding Basics

- **Written in modern C++, uses the STL:**
  - Particularly the vector, set, and map classes

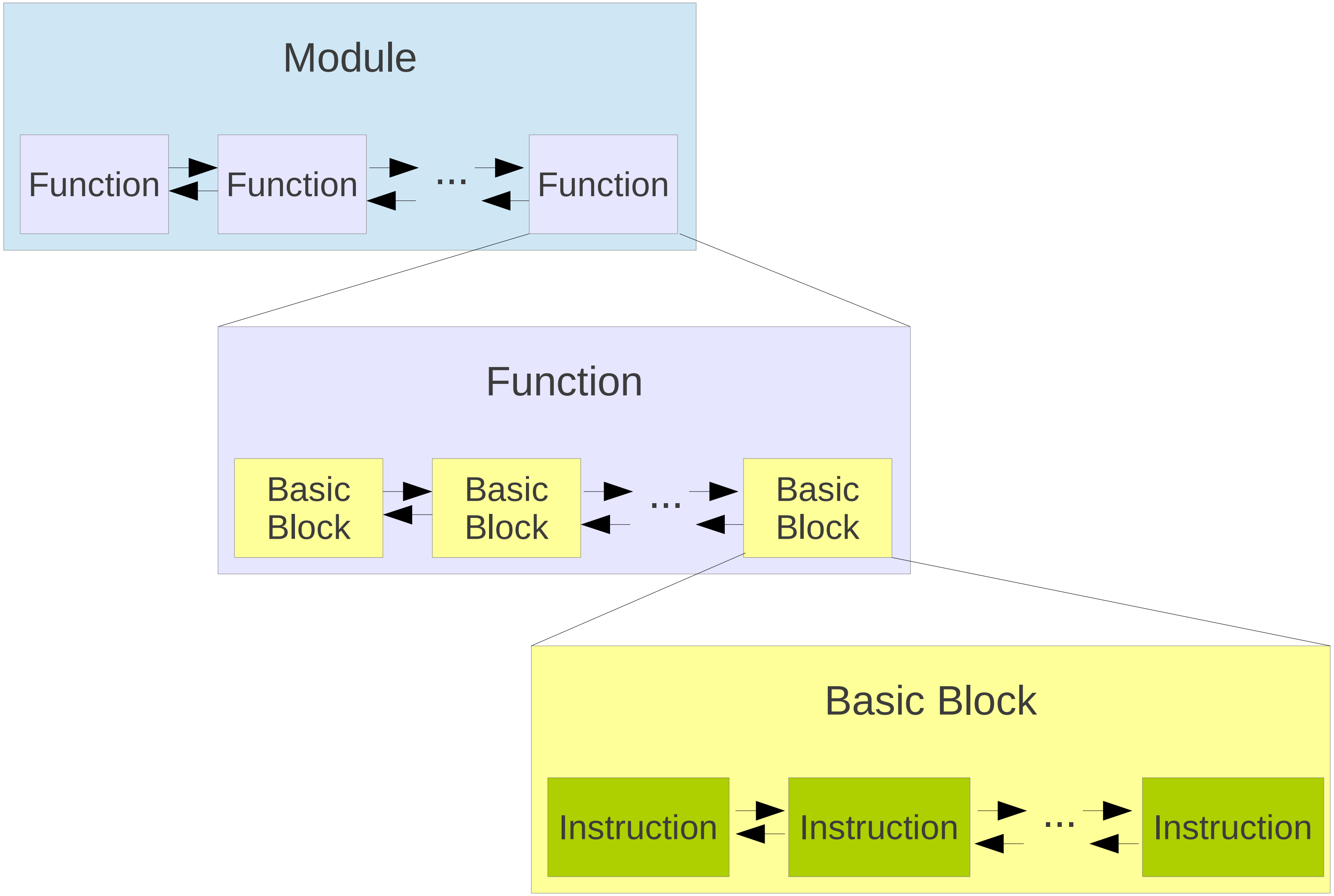
- **LLVM IR is almost all doubly-linked lists:**
  - **Module** contains lists of **Functions** & **GlobalVariables**
  - **Function** contains lists of **BasicBlocks** & **Arguments**
  - **BasicBlock** contains list of **Instructions**

- **Linked lists are traversed with iterators:**

```
Function *M = …

for (Function::iterator I = M->begin(); I != M->end(); ++I) {

  BasicBlock &BB = *I;

  ...
```

See also:
docs/ProgrammersManual.html

LLVM IR Structure



LLVM Pass Manager

- **Compiler is organized as a series of "passes":**
  - Each pass is one analysis or transformation
- **Four types of passes:**
  - ModulePass: general interprocedural pass
  - CallGraphSCCPass: bottom-up on the call graph
  - FunctionPass: process a function at a time
  - BasicBlockPass: process a basic block at a time
- **Constraints imposed (e.g. FunctionPass):**
  - FunctionPass can only look at "current function"
  - Cannot maintain state across functions

See also:
`docs/WritingAnLLVMPass.html`

Tutorial Overview

- **Introduction to the running example**
- **LLVM C/C++ Compiler Overview**
  - High-level view of an example LLVM compiler
- **The LLVM Virtual Instruction Set**
  - IR overview and type-system
- **Coding in LLVM**
- **Important LLVM Tools**
  - opt, code generator, JIT, test suite, bugpoint

LLVM tools

- **Basic LLVM Tools**
  - llvm-dis: Convert from .bc (IR binary) to .ll (human-readable IR text)
  - llvm-as: Convert from .ll (human-readable IR text) to .bc (IR binary)
  - opt: LLVM optimizer
  - llc: LLVM static compiler
  - llvm-link - LLVM bitcode linker
  - llvm-ar - LLVM archiver
- **Some Additional Tools**
  - bugpoint - automatic test case reduction tool
  - llvm-extract - extract a function from an LLVM module
  - llvm-bcanalyzer - LLVM bitcode analyzer
  - FileCheck - Flexible pattern matching file verifier
  - tblgen - Target Description To C++ Code Generator

See also:
http://llvm.org/docs/CommandGuide/

opt tool: LLVM modular optimizer

- **Invoke arbitrary sequence of passes:**
  - Completely control PassManager from command line
  - Supports loading passes as plugins from .so files

    opt -load foo.so -pass1 -pass2 -pass3 x.bc -o y.bc

- **Passes "register" themselves:**
  - **When you write a pass, you must write the registration**

```
RegisterPass<FunctionInfo> X("function-info",
        "15745: Function Information");
```