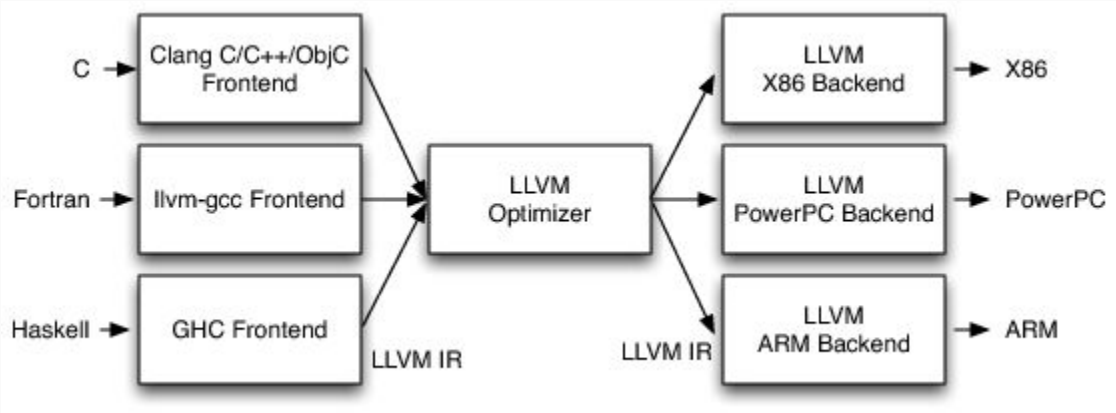
Overview of LLVM

Architecture of LLVM

Front-end: high-level programming language => LLVM IR

Optimizer: optimize/analyze/secure the program in the IR form

Back-end: LLVM IR => machine code



Optimizer

The optimizer’s job: analyze/optimize/secure programs.

Optimizations are implemented as passes that traverse some portion of a program to either collect information or transform the program.

A pass is an operation on a unit of IR code.

Pass is an important concept in LLVM.

LLVM IR

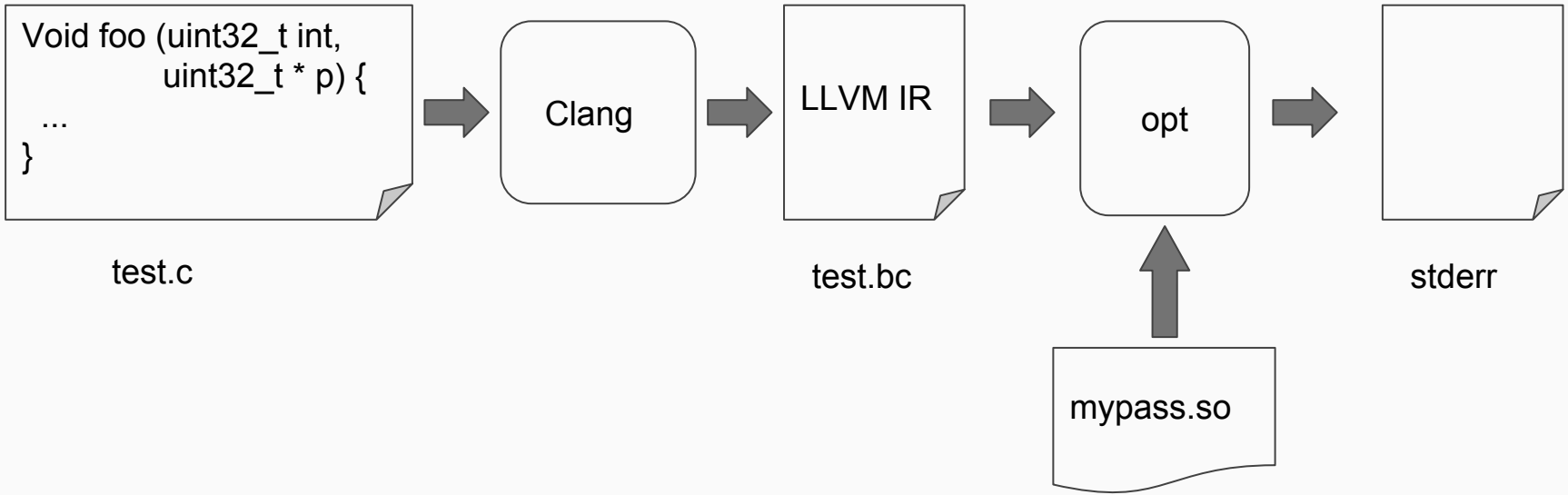
- A low-level **strongly-typed** language-independent, SSA-based representation.
- Tailored for static analyses and optimization purposes.

Part 1

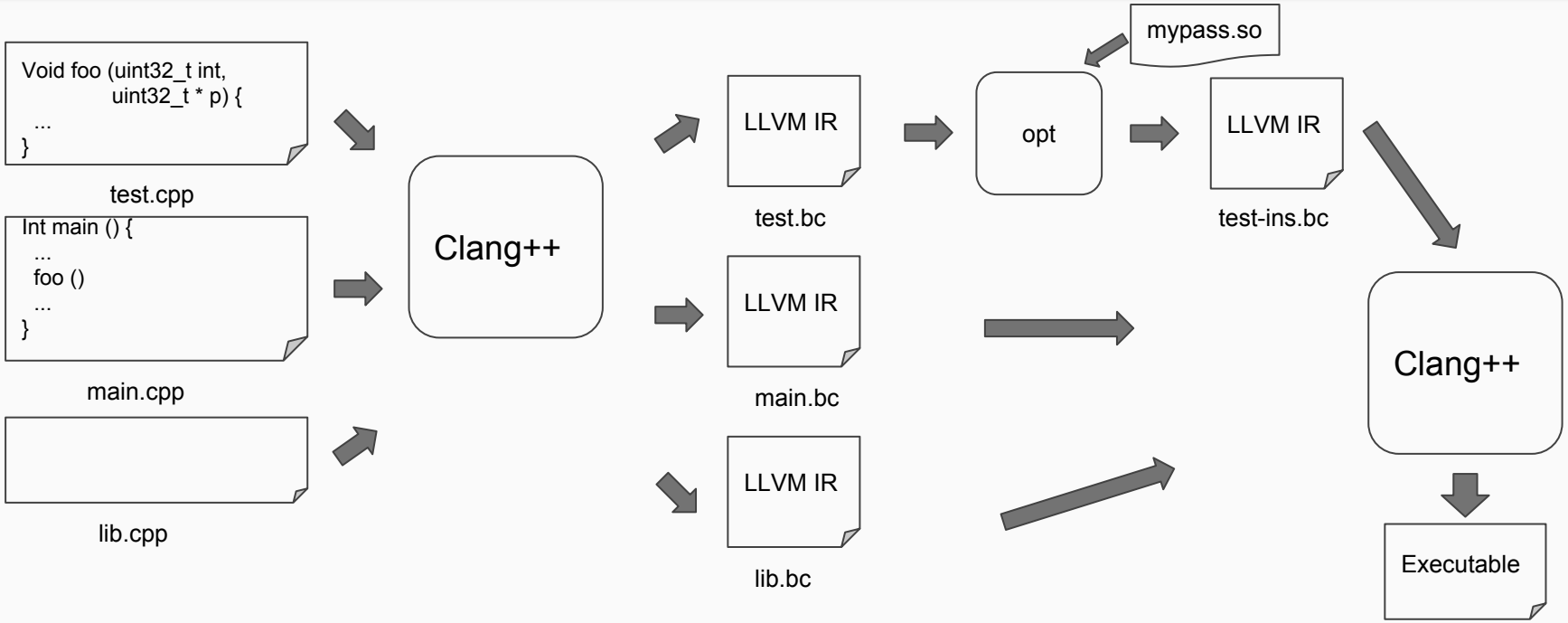
Part 1 has two kinds of passes:

- Analysis pass (section 1): only analyze code statically
- Transformation pass (section 2 & 3): insert code into the program

Analysis pass (Section 1)



Transformation pass (Section 2 & 3)



Section 1

Challenges:

- How to traverse instructions in a function

http://releases.llvm.org/3.9.1/docs/ProgrammersManual.html#iterating-over-the-instruction-in-a-function

- How to print to stderr

```
/// errs() - This returns a reference to a raw_ostream for standard error.
/// Use it like: errs() << "foo" << "bar";
raw_ostream &llvm::errs() {
```

Section 2 & 3

Challenges:

1. How to traverse basic blocks in a function and instructions in a basic block
2. How to insert function calls to the runtime library
    a. **Add the function signature to the symbol table of the module**

```
// getOrInsertFunction - Look up the specified function in the module symbol
// table.  If it does not exist, add a prototype for the function and return it.
// This version of the method takes a null terminated list of function
// arguments, which makes it easier for clients to use.
```

Module::getOrInsertFunction

Section 2 & 3

Challenges:

1. How to traverse basic blocks in a function and instructions in a basic block
2. How to insert function calls to the runtime library
    a. **Add the function signature to the symbol table of the module**

```
// Create the fib function and insert it into module M. This function is said
// to return an int and take an int parameter.
Function *FibF =
  cast<Function>(M->getOrInsertFunction("fib", Type::getInt32Ty(Context),
                                        Type::getInt32Ty(Context),
                                        nullptr));
```

Section 2 & 3

Challenges:

1. How to traverse basic blocks in a function and instructions in a basic block
2. How to insert function calls to the runtime library
    a. Add the function signature to the symbol table of the module
    b. Prepare argument for the function you want to call

Prepare arguments

Class ConstantInt: represent boolean and integral constants.

```
ConstantInt(IntegerType *Ty, const APInt& V);
```

Prepare arguments

Class ConstantInt: represent boolean and integral constants.

```
ConstantInt(IntegerType *Ty, const APInt& V);
```

Why do we need to specify Ty?

Prepare arguments

Class ConstantInt: represent boolean and integral constants.

```
ConstantInt(IntegerType *Ty, const APInt& V);
```

How to get Ty?

Prepare arguments

Class ConstantInt: represent boolean and integral constants.

```
ConstantInt(IntegerType *Ty, const APInt& V);
```

How to get Ty?

```
class Type
```

Prepare arguments

What if we need a pointer to an array of integers?

Prepare arguments

What if we need a pointer to an array of integers?

The easiest way to do it is to:

- Create a local constant array by using `class GlobalVariable`

Prepare arguments

What if we need a pointer to an array of integers?

The easiest way to do it is to:

- Create a local constant array by using `class GlobalVariable`

```
/// GlobalVariable ctor - This creates a global and inserts it before the
/// specified other global.
GlobalVariable(Module &M, Type *Ty, bool isConstant,
               LinkageTypes Linkage, Constant *Initializer,
               const Twine &Name = "", GlobalVariable *InsertBefore = nullptr,
               ThreadLocalMode = NotThreadLocal, unsigned AddressSpace = 0,
               bool isExternallyInitialized = false);
```

Prepare arguments

```
/// GlobalVariable ctor - This creates a global and inserts it before the
/// specified other global.
GlobalVariable(Module &M, Type *Ty, bool isConstant,
               LinkageTypes Linkage, Constant *Initializer,
               const Twine &Name = "", GlobalVariable *InsertBefore = nullptr,
               ThreadLocalMode = NotThreadLocal, unsigned AddressSpace = 0,
               bool isExternallyInitialized = false);
```

A GlobalVariable is a memory object, so it is always referred to by it address.

The type of an instance of GlobalVariable is pointer to its content (pointer to type Ty if the constructor above is used).

Prepare arguments

What if we need a pointer to an array of integers?

The easiest way to do it is to:

- Create a local constant by using `class GlobalVariable`
- Convert pointer to the correct type:
  - **LLVM Language Reference Manual** is your friend
  - Use LLVM API to insert useful instructions you found in the reference manual

    IRBuilder::Create[Instruction Name]

Section 2 & 3

Challenges:

1. How to traverse basic blocks in a function and instructions in a basic block
2. How to insert function calls to the runtime library
    a. Add the function signature to the symbol table of the module
    b. Prepare argument for the function you want to call
    c. Insert function calls

Insert function calls

1. Where to insert:
    a. Need to set up the insert point
    b. Take advantage of /solution/opt
2. How to insert: IRBuilder::CreateCall

Tips

- Take advantage of /solution/opt
- Take advantage of office hours
- Start early