# Building Programming Language Infrastructure with LLVM Components

Sunho Kim, Vassil Vassilev, and Lang Hames

# First steps

## What you'll need

- A build machine (a high end laptop or desktop should do)
- `git` for version control (LLVM is hosted on GitHub)
- `cmake` for configuring the build system
- `ninja` for building (much faster than make!)
- `clang`, `gcc`, or `MSVC` for compiling
- `lld` preferred for linking on Linux (binutils `ld` works, but is slow)
- Your trusty command line interface and editor

# First steps

## Checking out LLVM

- Check out llvm-project from github:

```
% git clone https://github.com/llvm/llvm-project.git
```

- Set up a build directory:

```
% cd llvm-project
% mkdir build
% cd build
```

# First steps

## Configuring LLVM

```
% cmake \
    -GNinja \
    -DCMAKE_BUILD_TYPE=RelWithDebInfo \
    -DLLVM_ENABLE_PROJECTS="clang" \
    -DLLVM_ENABLE_RUNTIMES="compiler-rt" \
    -DLLVM_TARGETS_TO_BUILD="AArch64;X86;NVPTX" \
    -DLLVM_PARALLEL_LINK_JOBS=1 \
    /path/to/llvm-project/llvm
```

# First Steps
## Building LLVM

```
% ninja clang clang-repl opt
```

# The LLVM Project

## Workshop Material

- LLVM Website: https://llvm.org
- Clang Website: https://clang.llvm.org
- LLVM C++ APIs change over time, but in-tree tutorial code is kept up-to-date
- Check LLVM docs, examples, and tutorial code for the latest versions
- The LLVM forums and LLVM discord are great places to go with questions

# The LLVM Project

## What is it?

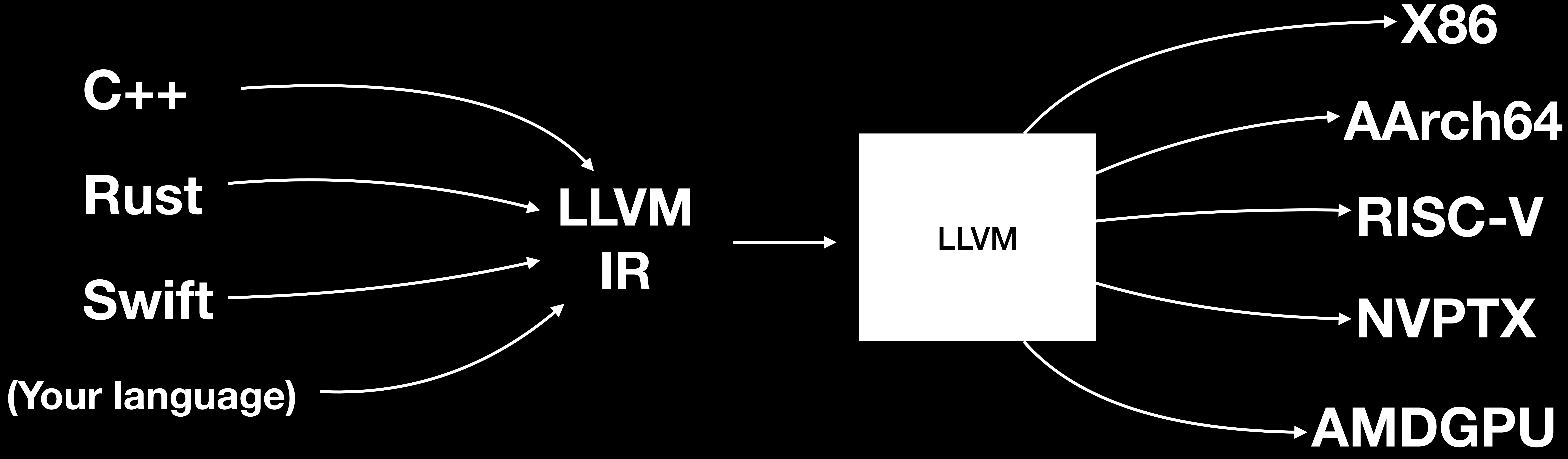
- Libraries for building programming language tools
    - Optimizers, code generators, assemblers, runtimes and standard libraries, debuggers, linkers, language frontends, JIT APIs, …
- Centered on the LLVM IR (intermediate representation)

# The LLVM Project

## LLVM IRs

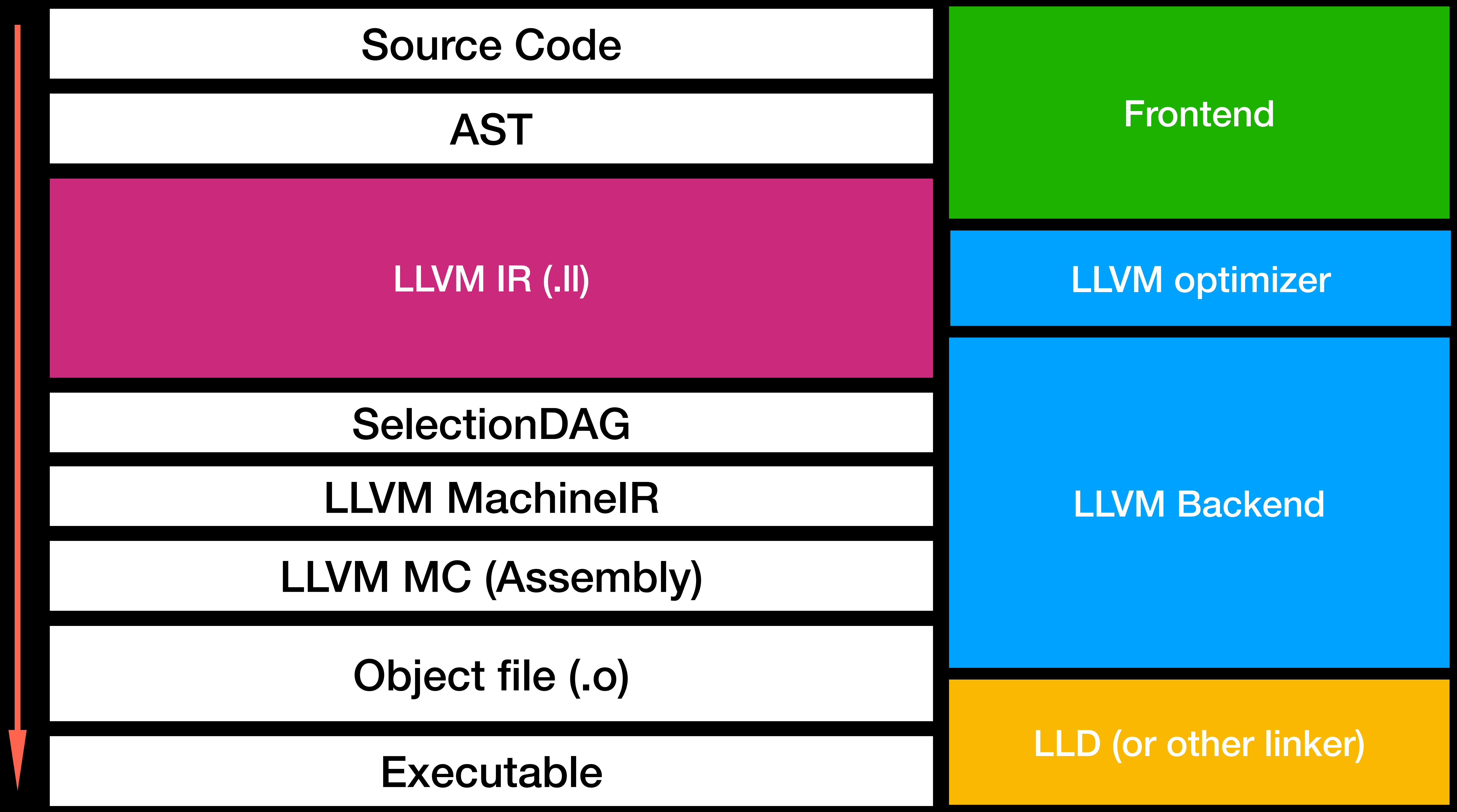
- LLVM’s target-independent “virtual instruction set”
- Generic, low-level, typed, SSA-form IR suitable as a target for front-ends
  - LLVM user writes codegen that emits LLVM IRs
- LLVM’s optimizer and back-end operate on input IRs to generate efficient machine code specific to the target architecture

# LLVM IR
## Benefits



# LLVM Pipeline

## Overview



# LLVM IR
## Hello, world!

```
@str = constant [13 x i8] c"Hello, world\00"

declare i32 @printf(i8*)

define i32 @main() {
  call i32 @printf(ptr @str)
  ret i32 0
}
```

```
% clang -o helloworld main.ll
```

# LLVM IR

## Example LLVM IR program

```c
int triangularNumber(int n) {
  return n*(n+1)/2;
}
```

```
define i8 @triangularNumber(i8 %n) {
  %add = add i8 %n, 1          // %add = n+1
  %mul = mul i8 %n, %add       // %mul = n * %add
  %res = sdiv i8 %mul, 2       // %res = %mul / 2
  ret i8 %res                  // return %res
}
```

Function

Instruction

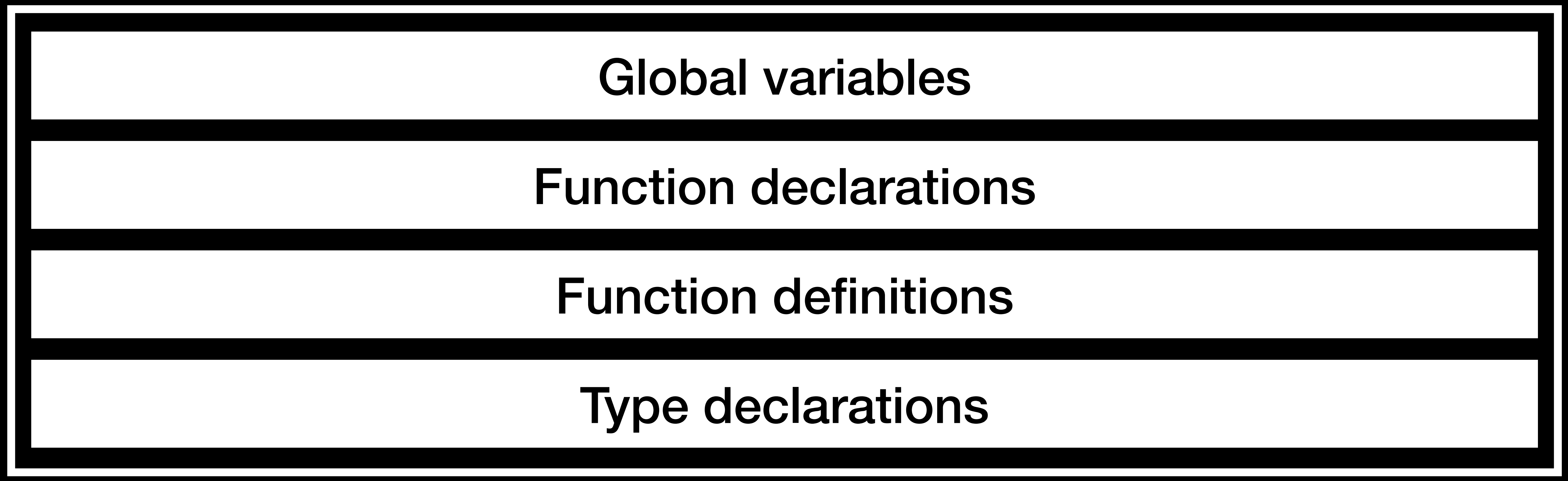
**SSA:** once you assign, you can't change later (e.g. you can't assign to %add another time)

**Explicit types:** type must be specified everywhere

# LLVM IR
## Module

- Top-level structure in LLVM
- Translation unit equivalent (each module can be compiled to individual .o file)
- Global values: functions and global variables

main.ll



# LLVM IR

## Global variable

main.ll

extern int gv

```
@gv = external global i32

define void @storeToGlobalVariable() {
  store i32 53, ptr @gv
  ret void
}
```

gv.ll

```
@gv = global i32 10
```

Linker → .elf .exe

```
% clang main.ll gv.ll
```

# LLVM IR

## Basic blocks

How to represent this code in LLVM IR?

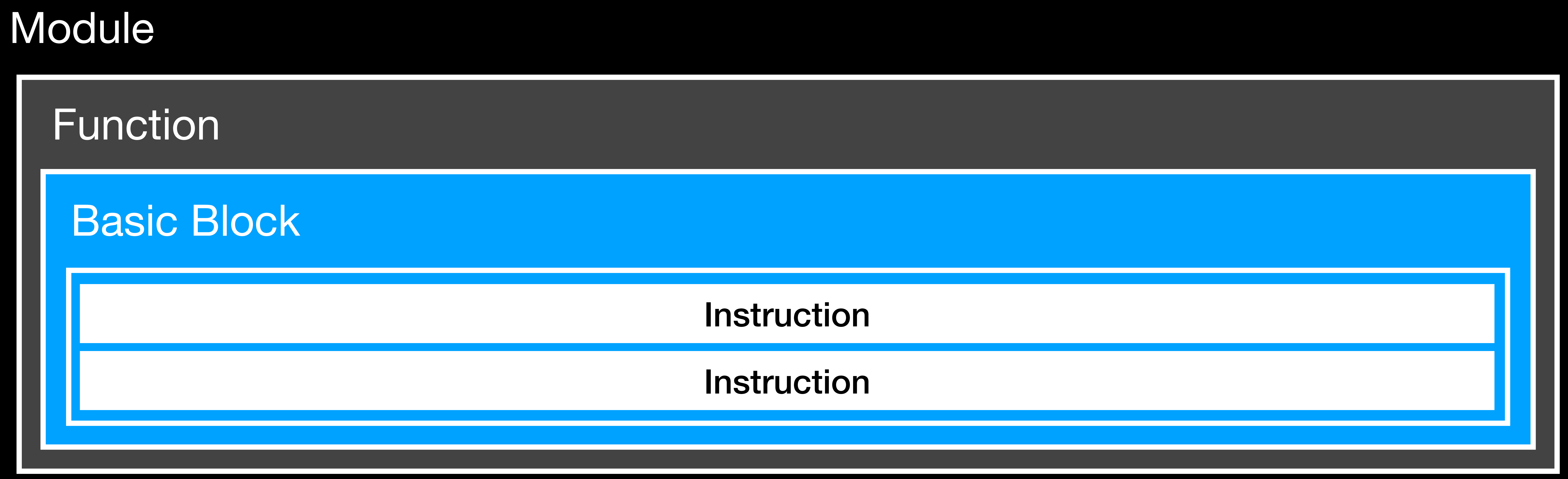
```
int fib(int n) {
  if (n <= 1) return n;
  return fib(n-1) + fib(n-2);
}
```

LLVM don’t have “if statement” syntax

# LLVM IR

## Basic blocks

- Basic building blocks of functions
- Block of instructions that ends with “terminal” instruction which is either branch or return



# LLVM IR

## Basic blocks

```
int fib(int n) {
  if (n<=1) return n;
  return fib(n-1) + fib(n-2);
}
```

```
entry:
  int result;
  if (n <= 1) goto ret_n;
```

N>1

N<=1

```
recursion:
  result = fib(n-1) + fib(n-2);
  goto end;
```

```
ret_n:
  result = n;
  goto end;
```

```
end:
  return result;
```

- Forms a directed graph of code blocks
- Within a single basic block, control flow is completely linear

# LLVM IR

## Fibonacci example

```
define i32 @fib(i32 %n) {
  %cmp = icmp sle i32 %n, 1    // %cmp = n <= 1
  br i1 %cmp, label %ret_n, label %recursion   // jump to %ret_n if %cmp else %recursion

ret_n:
  br label %end   // jump to %end

recursion:
  %n_1 = sub i32 %n, 1
  %fib_n_1 = call i32 @fib(i32 %n_1)  // %fib_n_1 = fib(n - 1)
  %n_2 = sub i32 %n, 2
  %fib_n_2 = call i32 @fib(i32 %n_2)  // %fib_n_2 = fib(n - 2)
  %add = add i32 %fib_n_1, %fib_n_2   // %add = %fib_n_1 + %fib_n_2
  br label %end   // jump to %end

end:
  %res = phi i32 [ %n, %ret_n ], [ %add, %recursion ]
  ret i32 %res    // return %res
}
```

%res = { %n if previous block was %ret_n
         %add if previous block was %recursion

# LLVM IR

## Tips

- Use clang to understand how idiomatic LLVM IR programs are structured and written
  - clang -emit-llvm main.cpp
- godbolt.com is our friend
  - Hover over a line of c++ source code, it will highlight corresponding part of LLVM IR program

# LLVM IR Optimization

- LLVM does its major optimizations on IR level; they optimize input IRs into more efficient one
- The power behind super efficient binary code generated by LLVM
- To name a few: constant folding, loop unrolling, dead code elimination, mem2reg, loop vectorization, …

# LLVM Passes and Analyses

- LLVM IR to IR transforms are called Passes
- Write your own passes to implement optimizations or analysis
- See LLVM doc “Using the New Pass Manager”

# LLVM IR Pass Example

## Optimize FFT

- Fast Fourier Transform (FFT): a powerful algorithm that can be used to do convolution operation within O(N log N) time
- Applications: signal processing, fast image blur filter, polynomial multiplication, convolutional neural network, …

# LLVM IR Pass Example

## Optimize FFT

- One of FFT implementations: Number Theoretic Transform (NTT)
  - Uses primitive root of unity modulo as basis for FFT
  - In nutshell, it just uses tons of modulo operations (a % mod)
  - But, modulo operation is slow in cpu
  - NTT can be made about 1.75 times faster by optimizing modulo operations

# LLVM IR Pass Example

## Optimize FFT

Modular arithmetics

a+b becomes (a+b) % MOD

a*b becomes (a*b) % MOD

Integers keep in the range [0,MOD) -> a and b also in range [0,MOD)

# LLVM IR Pass Example

## Optimize FFT

Modular addition optimization

(a+b) % 3

a: [0,3) b: [0,3) → a+b: [0,6)

| a+b | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| (a+b)%3 | 0 | 1 | 2 | 0 | 1 | 2 |

= a+b-3

If a+b >= 3, subtract 3

# LLVM IR Pass Example

## Optimize FFT

Modular addition optimization

(a+b) % MOD $\longrightarrow$

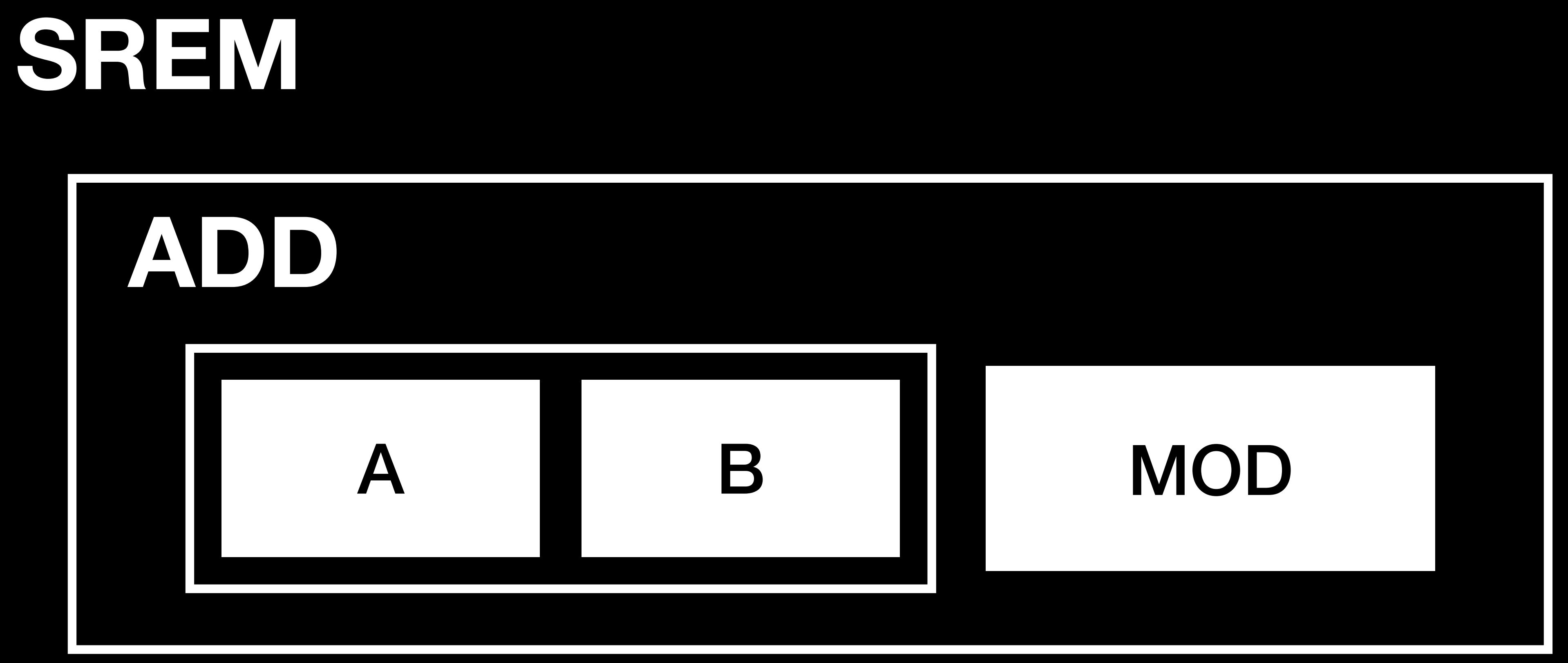
```
if (a+b >= MOD)
  a+b-MOD
else
  a+b
```

# LLVM IR Pass Example

## Optimize FFT

(a+b) % MOD

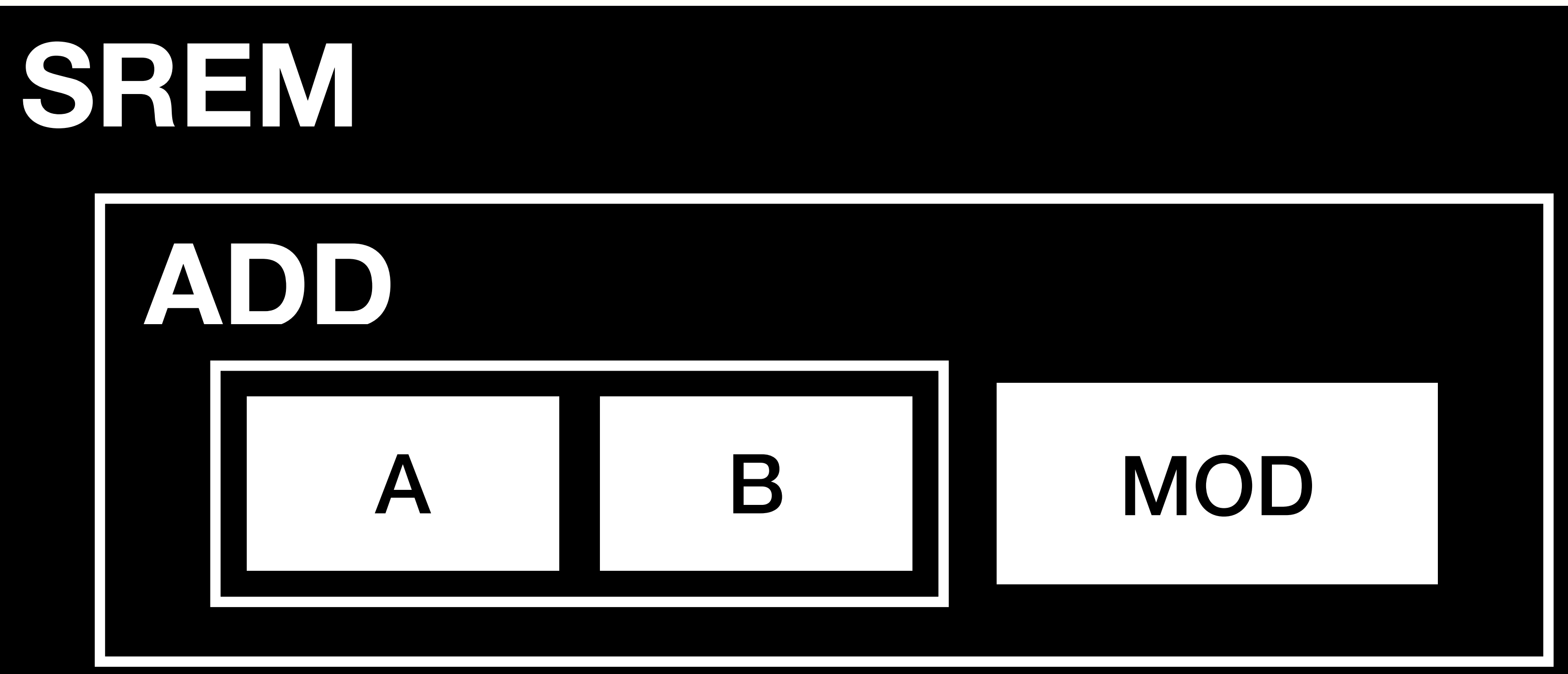
```
%add = add i32 %a, %b
%mod = load i32, ptr @mod
%res = srem i32 %add, %mod
```



# LLVM IR Pass Example

## Optimize FFT

```cpp
PreservedAnalyses run(Function &Func, FunctionAnalysisManager &) {
  for (auto& BasicBlock : Func) {
    for (auto& Inst : BasicBlock) {
      Value *A = nullptr, *B = nullptr, *Mod = nullptr;
      // Pattern match (A + B) % MOD
      if (match(&Inst, m_SRem(m_Add(m_Value(A), m_Value(B)), m_Value(Mod)))) {
        IRBuilder<> Builder(&Inst);
        auto* Add = Builder.CreateAdd(A, B);
        // Cmp = A + B >= MOD
        auto* Cmp = Builder.CreateICmp(ICmpInst::ICMP_UGE, Add, Mod);
        // Inst <- Cmp ? (A+B-MOD) : (A+B)
        Inst.replaceAllUsesWith(Builder.CreateSelect(Cmp, Builder.CreateSub(Add, Mod), Add));
      }
    }
  }
  return PreservedAnalyses::all();
}
```



# LLVM IR Pass Example

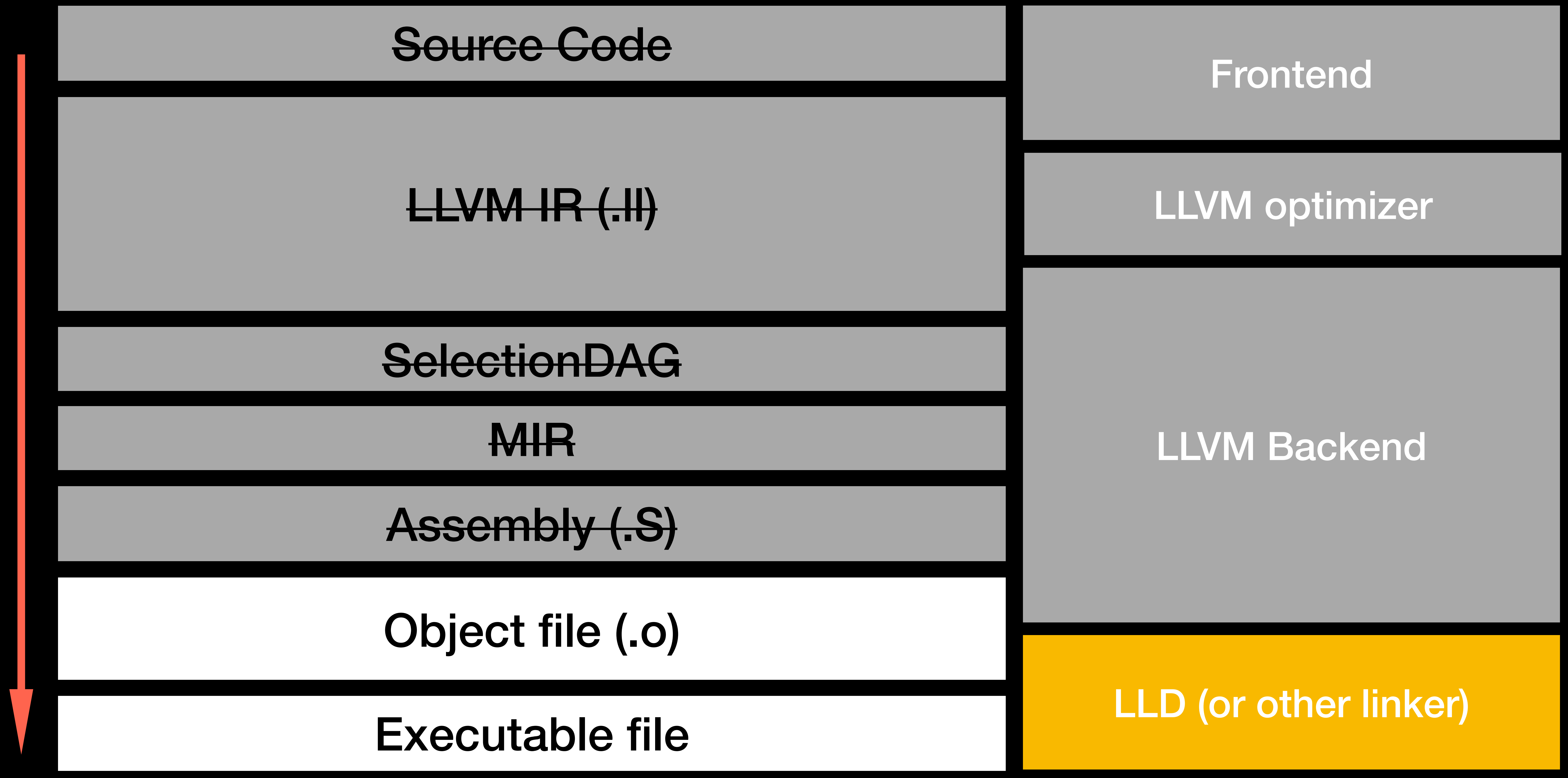
## Optimize FFT

- Alive 2 for IR transform verification
- https://alive2.llvm.org/ce/
- Actually run IRs to verify transform doesn't change the behavior
- https://reviews.llvm.org/D152568
- Built to battle undefined behavior such as integer overflow
  - e.g. verify transform doesn't introduce additional UB

# LLVM Backend

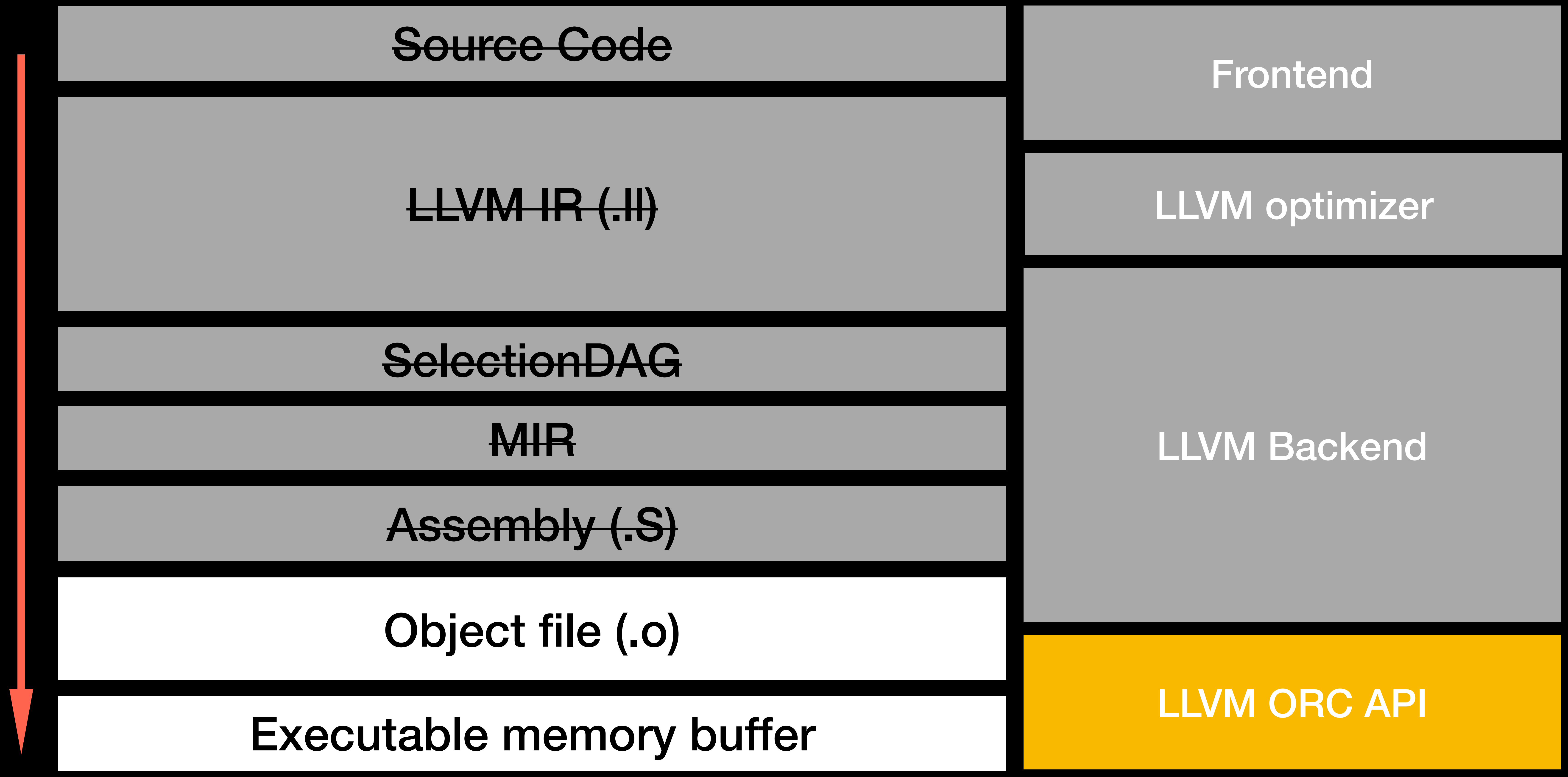
- Transform LLVM IR to actual machine code
- Has intermediate representations called SelectionDAG and MInst
- Uses TableGen language to pattern match DAG pattern and “lower” it to machine instructions
- Heuristic weight is given to patterns in order to select optimal instructions (e.g. select SIMD instruction when possible)
- It does its own instruction combine transforms in SelectionDAG level

# LLVM JIT API



~~Source Code~~
~~LLVM IR (.ll)~~
~~SelectionDAG~~
~~MIR~~
~~Assembly (.S)~~
Object file (.o)
Executable file
Frontend
LLVM optimizer
LLVM Backend
LLD (or other linker)

# LLVM JIT API


~~Source Code~~
~~LLVM IR (.ll)~~
~~SelectionDAG~~
~~MIR~~
~~Assembly (.S)~~
Object file (.o)
Executable memory buffer
Frontend
LLVM optimizer
LLVM Backend
LLVM ORC API

# LLVM JIT APIs

# Today's Workshop

- We're going to build a Read-Eval-Print Loop (REPL) for *Kaleidoscope*
- Kaleidoscope is a simple procedural language used in the LLVM tutorials:

```
def fib(n)
  if n < 3 then
    1
  else
    fib(n - 1) + fib(n - 2)
```

- We'll take the language for granted and focus on the JIT implementation
- We're going to start out eagerly compiling, and make it progressively lazier

# Related materials

- LLVM Kaleidoscope tutorial series
- LLVM Example programs
- LLVM Documentation
- LLVM Developer Meeting Talks
  - 2018 — Updating ORC JIT for concurrency (ORCv2, which we'll use today)
  - 2021 — ORC deep-dive, covering JITLink & the ORC runtime
  - 2016 — Original ORC introduction (mostly outdated)

# ORC — On Request Compilation

## LLVM’s “JIT” APIs

- A library for building JITs and JIT-like things
- Contains a Just-In-Time Linker that can patch in relocatable object files
- Makes it easy to plug in your own compiler(s)
- Supports lazy compilation, concurrent compilation, and remote execution
- Directly inspect and modify JIT’d machine code
- Construct programs by combining compiler inputs (rather than linking compiler outputs), then let ORC trigger compilations as needed

# Kaleidoscope
## Exercises

1. Basic REPL loop, Kaleidoscope compiled up-front
2. Defer compilation until first lookup using a custom `MaterializationUnit`
3. Defer lookup until runtime using *lazy re-exports*
4. Make precompiled symbols accessible to JIT’d code
5. Use `ObjectLinkingLayer::Plugin` to access JIT’d objects during linking

# Workshop Code

## Checking out, configuring, and building the example code

```
% git clone https://github.com/compiler-research/pldi-tutorials-2023.git
% cd pldi-tutorials-2023
% mkdir build && cd build
% cmake -GNinja -DCMAKE_BUILD_TYPE=Debug \
    -DLLVM_DIR=/path/to/llvm-build \
    ..
% ninja
```

# Workshop Code

## Checking out, configuring, and building the example code

```
% git clone https://github.com/compiler-research/pldi-tutorials-2023.git
% cd pldi-tutorials-2023
% mkdir build && cd build
% cmake -GNinja -DCMAKE_BUILD_TYPE=Debug \
    -DLLVM_DIR=/path/to/llvm-build/lib/cmake/llvm \
    ..
% ninja
```

# Kaleidoscope.h

Parser and JIT Definition

# Kaleidoscope.h

## Overview

- Kaleidoscope `FunctionAST` definition
  - Just a pair of `PrototypeAST` + `ExprAST`
  - Kaleidoscope functions are compile units for the purpose of this tutorial
- `KaleidoscopeParser` — Incremental Kaleidoscope parser
  - `parse` — Translate source to `FunctionAST` [ + expression name ]
  - `codegen` — Compile `FunctionAST` to `llvm::ThreadSafeModule`

# Kaleidoscope.h

## KaleidoscopeJIT — member variables

- `ExecutionSession` — Top-level management for ORC programs
  - Create JITDylibs, manage JIT'd resources, perform symbol lookup
- `DataLayout` — LLVM IR data lowering: endianness, alignment, mangling
- `Mangler` — Map C/IR symbol names to linker-level names suitable for lookup

# ORC APIs

## ORC uses *linker-mangled* symbol names

What are linker-mangled names?

E.g. on Darwin, C `void foo(void)`
becomes assembly symbol `_foo`

Why use linker-mangled names in a JIT?

Consistency with existing compilers and compiled code

# Kaleidoscope.h

## KaleidoscopeJIT — member variables

- `ExecutionSession` — Top-level management for ORC programs
  - Create JITDylibs, manage JIT'd resources, perform symbol lookup
- `DataLayout` — LLVM IR data lowering: endianness, alignment, mangling
- `Mangler` — Map C/IR symbol names to linker-level names suitable for lookup
- `ObjectLinkingLayer` — Links relocatable object files
- `IRCompileLayer` — Compile LLVM IR to relocatable object files
- `JITDylib` — The program representation container

# Kaleidoscope.h

## KaleidoscopeJIT — methods

- `Create` tries to create members, bails out on error
  - `ExecutorProcessControl` abstracts the target process (in this case the current process) — it's owned by `ExecutionSession`
  - `JITTargetMachineBuilder` builds `llvm::TargetMachine` instances for LLVM IR compilation
- `~KaleidoscopeJIT` destructor calls `ExecutionSession::endSession`
  - Releases JIT'd resources
- `KaleidoscopeJIT` constructor is trivial — just sets member variables

# Exercise 1
# Simple REPL, Eager Compilation

# Exercise 1

## Includes

```
#include "Kaleidoscope.h"

#include "llvm/LineEditor/LineEditor.h"
#include "llvm/Support/InitLLVM.h"
#include "llvm/Support/TargetSelect.h"
```

This workshop

LLVM Line Editor,
LLVM Initialization,
and Target Initialization

# Exercise 1

## Initialization

```
int main(int argc, char *argv[]) {
  InitLLVM X(argc, argv);

  InitializeNativeTarget();
  InitializeNativeTargetAsmPrinter();
  InitializeNativeTargetAsmParser();

  ExitOnError ExitOnErr("kaleidoscope: ");
```

Initialize LLVM, initialize native target, create `ExitOnErr`

# Exercise 1

## JIT, Parser, and REPL setup

```
std::unique_ptr<KaleidoscopeJIT> J =
    ExitOnErr(KaleidoscopeJIT::Create());

KaleidoscopeParser P;

llvm::LineEditor LE("kaleidoscope-repl");
while (auto Line = LE.readLine()) {
  auto ParseResult = P.parse(*Line);
  if (!ParseResult)
    continue;
  …
```

JIT

Parser

Parse Input to Kaleidoscope AST

# Exercise 1

## Compile Kaleidoscope AST to LLVM IR, add to JIT

```
auto IRMod =
  P.codegen(std::move(ParseResult->FnAST), J->DL);

if (!IRMod)
  continue;

ExitOnErr(
    J->CompileLayer.add(J->MainJD, std::move(*IRMod)));
```

Generate LLVM IR Module from Function AST

Add IR Module to the JIT

# Exercise 1

## Check for top-level expression, lookup

Lookup triggers
LLVM IR compilation

```
if (ParseResult->TopLevelExpr.empty())
  continue;

Expected<ExecutorSymbolDef> ExprSym = J->ES->lookup(
    &J->MainJD, J->Mangle(ParseResult->TopLevelExpr));

if (!ExprSym) {
  errs() << "Error: "
         << toString(ExprSym.takeError()) << "\n";
  continue;
}
```

Print errors and continue

# Exercise 1

## Cast expression address to pointer, execute

Cast compiled address to pointer

```
double (*Expr)() =
    ExprSym->getAddress().toPtr<double (*)()>();

double Result = Expr();
```

Call JIT’d code

```
outs() << "Result = " << Result << "\n";
```

# Exercise 1

## First JIT'd Kaleidoscope code

```
% ./bin/p2-ex1
kaleidoscope> def add(a b) a + b;
kaleidoscope> add(1, 2);
Result = 3.000000e+00
```

# Try un-commenting:

```
// dbgs() << "Compiling " << ParseResult->FnAST->getName() << "\n";
auto IRMod = P.codegen(std::move(ParseResult->FnAST), J->DL);
```

# Exercise 1

## First JIT'd Kaleidoscope code (with debugging output)

```
% ./bin/p2-ex1
kaleidoscope> def add(a b) a + b;
Compiling add
kaleidoscope> add(1, 2);
Compiling expr.0
Result = 3.000000e+00
```

# Exercise 2
# Wrapping Kaleidoscope ASTs

# Exercise 2

## Wrapping custom program representations

- Subclass `MaterializationUnit` to wrap custom program representations
- Construct with linker-level interface that would be produced if compiled
- Implement `materialize` method — called if lookup matches interface
- Implement `discard` method — called if interface symbol is overridden

# Exercise 2

## MaterializationResponsibility

- `materialize` will be passed a `MaterializationResponsibility` object
- Initially responsible for materializing all symbols, even if only some requested
- To continue materializing all symbols just pass the object along, otherwise…
- Use `getRequestedSymbols` to identify the symbols that are actually needed
- Partition program representation, return unrequested symbol with `replace`
- Continue materialization with requested symbols
- See 2018 LLVM Developer Meeting JIT talk for more details

# Exercise 2

## Subclass MaterializationUnit

Subclass
MaterializationUnit

```
class KaleidoscopeASTMU : public MaterializationUnit {
public:
  KaleidoscopeASTMU(KaleidoscopeParser &P,
                    KaleidoscopeJIT &J,
                    std::unique_ptr<FunctionAST> FnAST)
    : MaterializationUnit(getInterface(J, *FnAST)),
      P(P), J(J), FnAST(std::move(FnAST)) {}
```

# Exercise 2

## The materialize method

Responsibility
Object

```
void materialize(
    std::unique_ptr<MaterializationResponsibility> R) {
  if (auto IRMod = P.codegen(std::move(FnAST), J.DL))
    J.CompileLayer.emit(std::move(R), std::move(*IRMod));
  else
    R->failMaterialization();
}
```

On success, pass IRMod
and Responsibility object
along to CompileLayer

If something goes wrong
we must explicitly report it

# Exercise 2

## Constructing the interface

Map symbol name to flags

```
static MaterializationUnit::Interface
getInterface(KaleidoscopeJIT &J, FunctionAST &FnAST) {
  SymbolFlagsMap Symbols;
  Symbols[J.Mangle(FnAST.getName())] =
      JITSymbolFlags::Exported | JITSymbolFlags::Callable;
  return { std::move(Symbols), nullptr };
}
```

Return flags map
(and initializer symbol,
always null in our case)

# Exercise 2

## Old approach: Eagerly compile, add LLVM IR

```
dbgs() << "Compiling " << ParseResult->FnAST->getName()
       << "\n";
auto IRMod =
  P.codegen(std::move(ParseResult->FnAST), J->DL);
…
ExitOnErr(
  J->CompileLayer.add(J->MainJD, std::move(*IRMod)));
```

# Exercise 2

## New approach: add ASTs

```
ExitOnErr(J->MainJD.define(
    std::make_unique<KaleidoscopeASTMU>(
        P, *J, std::move(PR->FnAST))));
```

# Try un-commenting:

```
// dbgs() << "Compiling " << ParseResult->FnAST->getName() << "\n";
if (auto IRMod = P.codegen(std::move(ParseResult->FnAST), J->DL))…
```

# Exercise 2

## Getting lazier…

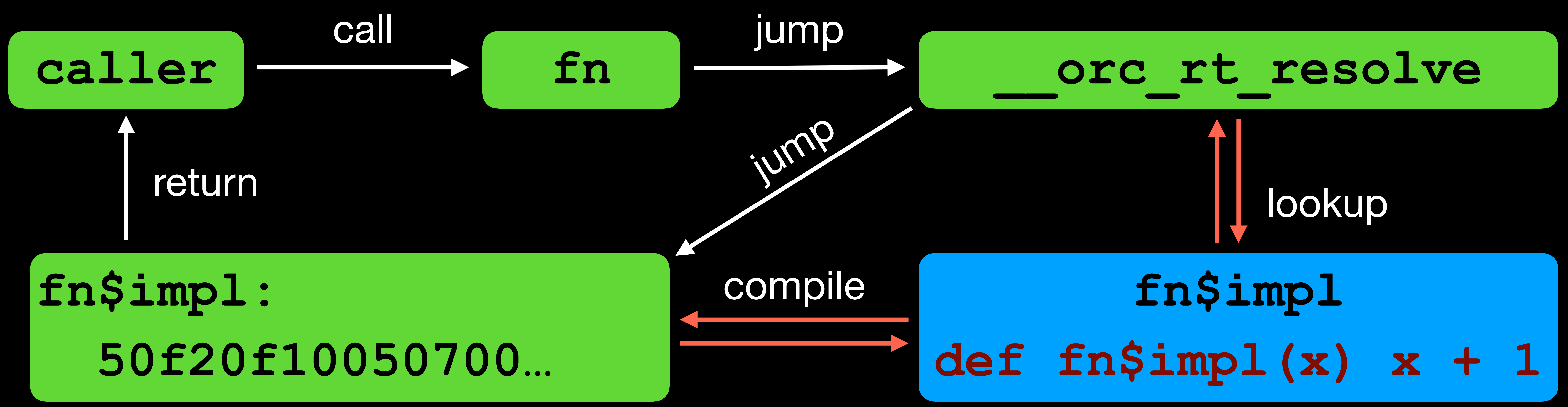
```
% ./bin/p2-ex2
kaleidoscope> def neg(x) 0 - x;
kaleidoscope> def abs(x) if x < 0 then neg(x) else x;
kaleidoscope> abs(3);
Compiling expr.0
Compiling abs
Compiling neg
Result = 3.000000e+00
```

# Exercise 3
# Lazy re-exports

# Exercise 3

## Lazy compilation using lazy re-exports

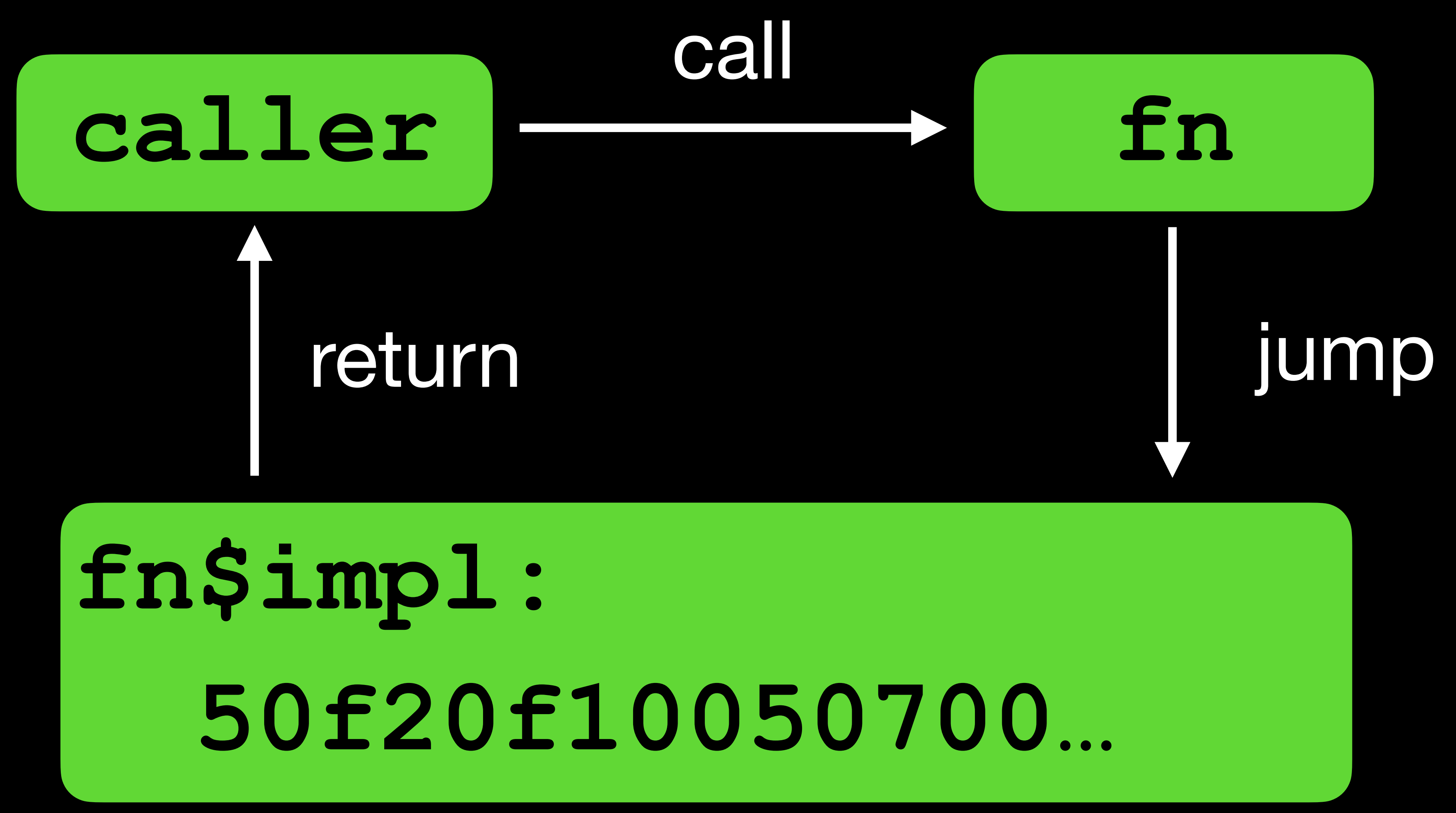
- Create lazy re-exports from a map of stub names to function body names
- ORC generates the stubs for you
- E.g. for `"fn"` → `"fn$impl"`:



# Exercise 3

## Lazy compilation using lazy re-exports

- Create lazy re-exports from a map of stub names to function body names
- ORC generates the stubs for you
- E.g. for `"fn"` → `“fn$impl"`:




After first call the stub jumps directly to the compiled body

# Exercise 3

## Lazy re-exports initialization

```
auto EPCIU =
  ExitOnErr(EPCIndirectionUtils::Create(*J->ES));

auto EPCIUCleanup = make_scope_exit([&]() {
  if (auto Err = EPCIU->cleanup())
    J->ES->reportError(std::move(Err));
});

auto &LCTM = EPCIU->createLazyCallThroughManager(*J->ES,
    ExecutorAddr::fromPtr(&handleLazyCompileFailure));

ExitOnErr(setUpInProcessLCTMReentryViaEPCIU(*EPCIU));

auto ISM = EPCIU->createIndirectStubsManager();
```

BOILERPLATE !

# Exercise 3

## Create re-exports map, rename function body

```
std::string FnImplName = PR->FnAST->getName() + "$impl";
SymbolAliasMap ReExports({
  { J->Mangle(PR->FnAST->getName()),
    { J->Mangle(FnImplName),
      JITSymbolFlags::Exported | JITSymbolFlags::Callable }
    }
});

PR->FnAST->setName(std::move(FnImplName));
```

# Exercise 3

## Add function body and lazy reexports to JIT

Add `fn$impl` definition

```
ExitOnErr(J->MainJD.define(
  std::make_unique<KaleidoscopeASTMU>(
    P, *J, std::move(PR->FnAST))));

ExitOnErr(J->MainJD.define(
  lazyReexports(LCTM, *ISM, J->MainJD,
                std::move(ReExports))));
```

Defines `fn` stub, which will lookup and run `fn$impl` if called

# Exercise 3

## Lazy compilation using lazy re-exports

- In Exercise 2 we deferred AST compilation until lookup
- In this exercise we deferred lookup of the function body until runtime
- Together they defer compilation of the function body until runtime
- Custom `MaterializationUnit` + `lazyReexports` provides lazy compilation for arbitrary program representations
- Lookup is thread-safe — by building laziness on top of it we inherit that safety

# Exercise 3

## Laziest…

```
% ./bin/p2-ex3
kaleidoscope> def neg(x) 0 - x;
kaleidoscope> def abs(x) if x < 0 then neg(x) else x;
kaleidoscope> abs(3);
Compiling expr.0$impl
Compiling abs$impl
Result = 3.000000e+00
kaleidoscope> abs(0 - 3);
Compiling expr.1$impl
Compiling neg$impl
Result = 3.000000e+00
```

# Exercise 4
# Reflecting Precompiled Symbols

# Exercise 4

## Reflecting Precompiled Symbols

- We'll use a `DefinitionGenerator` to add symbols in response to lookups
  - Generators are attached to JITDylibs
  - Lookups that don't match existing symbols will fall through to generators
  - Generator for this exercise uses `dlsym` / `GetProcAddress` under the hood
- Generators have high overhead — avoid if possible, use sparingly
- To expose a fixed set of precompiled symbols use `absoluteSymbols` instead: it's faster, and prevents unintended symbols from being pulled in

# Exercise 4

## Create and link against a “process-symbols” JITDylib

Create JITDylib to
hold process symbols

```
auto &ProcessSymbolsJD =
  J->ES->createBareJITDylib("<Process_Symbols>");

ProcessSymbolsJD.addGenerator(ExitOnErr(
  EPCDynamicLibrarySearchGenerator::
    GetForTargetProcess(*J->ES)));

J->MainJD.addToLinkOrder(ProcessSymbolsJD);
```

Add a Process
Symbols Generator

Link MainJD against Process Symbols JITDylib

# Exercise 4

## Add a precompiled function

extern “C” gives us C / IR names,
which line up with Kaleidoscope

```
extern "C" double circleArea(double radius) {
  return M_PI * radius * radius;
}
```

# Exercise 4

## Use from Kaleidoscope

```
% ./bin/p2-ex4
kaleidoscope> extern circleArea(r);
kaleidoscope> def cylinderVolume(r h) circleArea(r) * h
kaleidoscope> cylinderVolume(1, 2);
Result = 6.283185e+00
```

# Exercise 5
# Object Linking Layer Plugins

# Exercise 5

## The JIT Linker

- JITLink links relocatable object files directly into memory
- It has a public API and core data structure — the `LinkGraph`
- You can install custom plugins to inspect / mutate LinkGraphs during linking
  - Plugins can add passes to the linker pipeline in specific phases:
    before pruning, before and after allocation, and before and after fixups
  - Passes can inspect / mutate sections, symbols, and relocations
    (subject to the rules of the phase they run in)
- See the LLVM 2021 ORC Deep-dive talk on YouTube for more info

# Exercise 5

## Custom Plugin Class

```
class MyPlugin : public ObjectLinkingLayer::Plugin {
public:
  void modifyPassConfig(
      MaterializationResponsibility &MR, LinkGraph &G,
      PassConfiguration &PassConfig) override {
    PassConfig.PostAllocationPasses.push_back(
        [this](LinkGraph &G) { return printGraph(G); });
  }
…
```

# Exercise 5

## The printGraph pass

```
Error printGraph(LinkGraph &G) {
  // Print graph name:
  outs() << "Graph " << G.getName() << "\n";

  // Loop over sections
  for (auto &Sec : G.sections()) {
    outs() << "  Section " << Sec.getName() << ", "
           << Sec.getMemProt() << "\n";
  …
```

# Exercise 5

## Demo

```
% ./bin/p2-ex5
kaleidoscope> extern circleArea(r);
kaleidoscope> circleArea(1);
Graph m.0-jitted-objectbuffer
  Section __TEXT,__text, R-X
  Symbols:
    0x10dd54000: _expr.0$impl
  Blocks:
    0x10dd54000: content = { 0x50 0xf2 0x0f 0x10 0x05 0x07 0x00
0x00 0x00 0xe8 ... }, 16-bytes, 4 edges
  …
Result = 3.141593e+00
```

# Exercise 5

## What is this good for?

- ORC uses it to discover initializers, implement exceptions and TLV, etc.
- Apply range-based optimizations to instruction streams (on by default)
- Fine-grained control over the implementation of lazy control flow, e.g. …
  - Record and rewrite call-sites to bypass stub functions
  - Insert nop-sleds at the start of functions
- Insert some instrumentation and logging without polluting compiled objects

# Further topics

# Reoptimization

## Tiering up generated code performance

- Fast compile up-front at low optimization level
- Recompile hot functions at a higher optimization level
- ORC’s symbol lookup system makes this easy to express in a cross-platform and thread safe way
- Initial work is happening as part of this year’s Google Summer of Code

# Speculation

## Sometimes it pays to be a little less lazy

- If we wait until we hit a stub then execution of JIT'd code blocks on compiles
- Some function executions can be guessed at in advance
  - Use program analysis, or profile data, or both
- If we have free resources then kick compiles off early (it just takes a lookup)
- When we speculate effectively we should be able to hide compile latency
- Identifying good quality speculation opportunities is an open problem in LLVM

# Further, further topics

## There is a lot we couldn't cover today

- JIT Linker developments (new backends, optimizations)
- The ORC runtime (dlopen emulation)
- Remote execution
- Debugger and profiling support for JIT'd code
- …
- For background check out LLVM tutorials, examples, Dev Talks, etc.
- For open problems check out LLVM GitHub issues
- Join the LLVM community discourse forums and discord channel

# Questions?

# Part 3. Compiler-As-A-Service

## How to use Clang as a library. Incremental compilation. How to build a C++ interpreter. C/C++/Python on-demand interoperability.

Vassil Vassilev

# The Clang Project
## What is it?

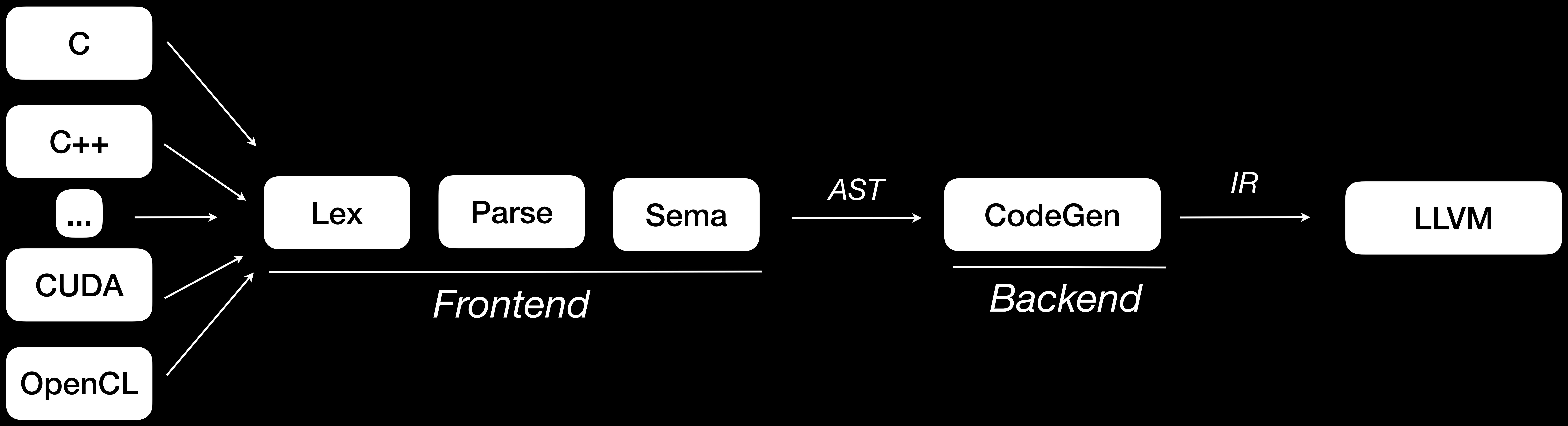
- Clang is a compiler which supports C, C++, Objective-C, and Objective-C++ programming languages, as well as the OpenMP, OpenCL, RenderScript, CUDA, SYCL, and HIP frameworks.
- Just like LLVM, Clang is built by a set of reusable components and can be used as a library.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Analysis | Basic | Lex | Sema | ... | Driver | Format | FrontendTool | Interpreter |
| AST | CodeGen | Parse | Index | | Edit | Frontend | StaticAnalyzer | Tooling |

# Going Beyond Kaleidoscope

## The Clang Compiler Infrastructure. Classic Data Flow

- Clang takes input pipes it through a frontend and a backend and produces machine code



# Exercise 1
# Using Clang As A Library

AST

Tooling

# Exercise 1

## p3-ex1.cpp: Extracting documentation from source

Run the compiler on given code.

```
#include "clang/AST/Comment.h"
#include "clang/AST/DeclTemplate.h"
#include "clang/Tooling/Tooling.h"
...
auto ASTU = tooling::buildASTFromCodeWithArgs(Code);
ASTContext &C = ASTU->getASTContext();
TranslationUnitDecl* TU = C.getTranslationUnitDecl();
TU->dump();
```

```
|-ClassTemplateDecl 0x7f83db895f48 input.cc:4:1 line:5:40 col:8 ComplexNumbe
| |-TemplateTypeParmDecl 0x7f83db895dd0 line:4:10 col:19 col:19
| |-CXXRecordDecl 0x7f83db895e98 line:5:1 col:40 col:8      ComplexNumber
| | ...
| | |-FullComment 0x7f83dc00c200 line:2:4 col:52
| | | `-ParagraphComment 0x7f83dc00c1d0 col:4 col:52
| | |   `-TextComment 0x7f83dc00c1a0 col:4 col:52
          Text=" This is the documentation for the ComplexNumber."
```

# Exercise 2
# A Simple C++ REPL

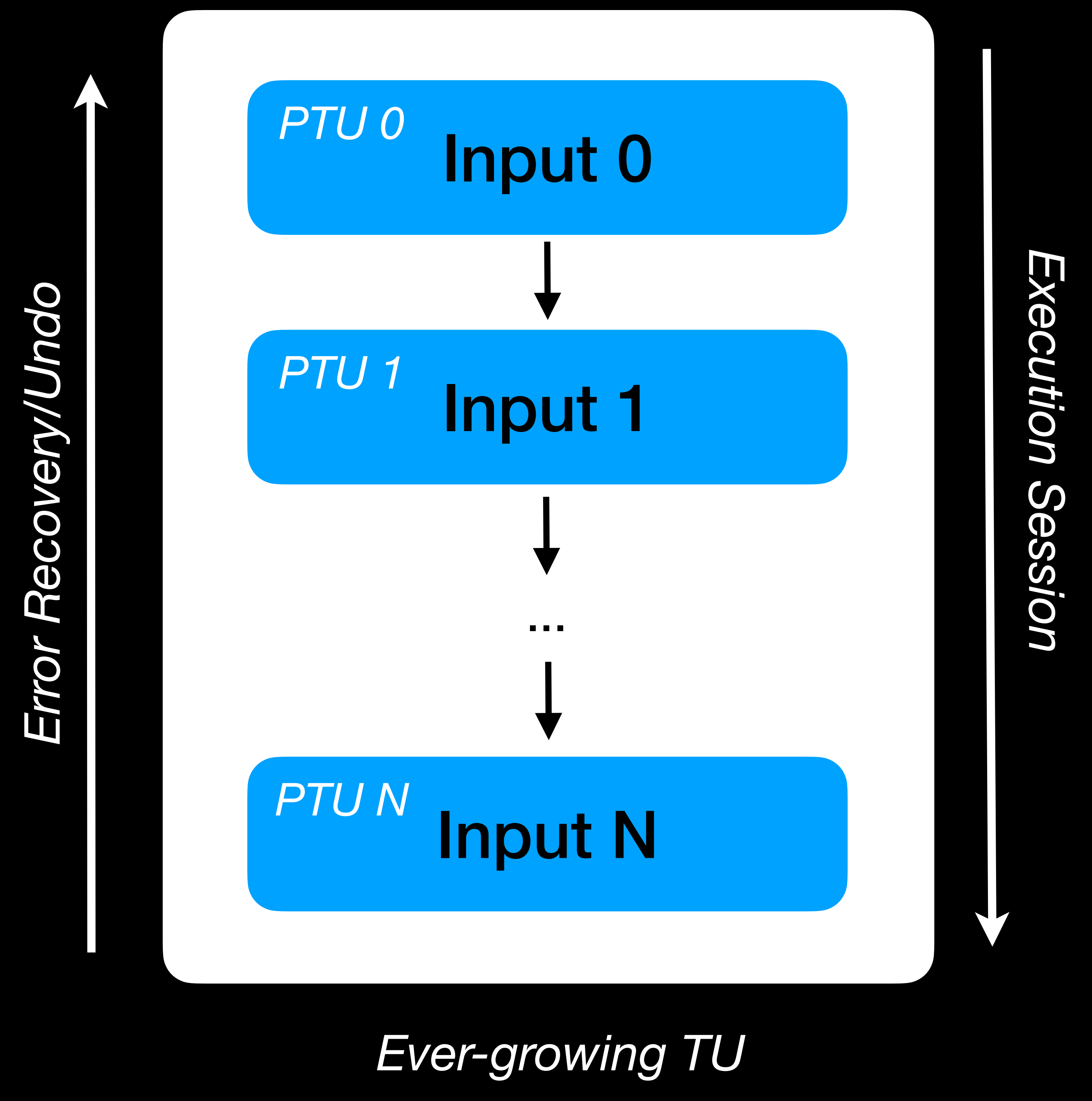
Frontend

Interpreter

# Exercise 2

## Incremental Compilation in Clang

- We can split the translation unit into a sequence of partial translation units (PTU)
- Processing a PTU might extend an earlier PTU (template instantiation



# Exercise 2

## p3-ex2.cpp: A Simple C++ REPL

-fplugin=my_plugin.so

Standard compiler flags

```
// Initialize our builder class.
clang::IncrementalCompilerBuilder CB;
CB.SetCompilerArgs({"-std=c++20"}); // pass `-xc` for C.

// Create the incremental compiler instance.
auto CI = ExitOnErr(CB.CreateCpp());

// Create the interpreter instance.
auto Interp = ExitOnErr(Interpreter::create(std::move(CI)));
```

Creates an incremental compiler instance

# Exercise 2

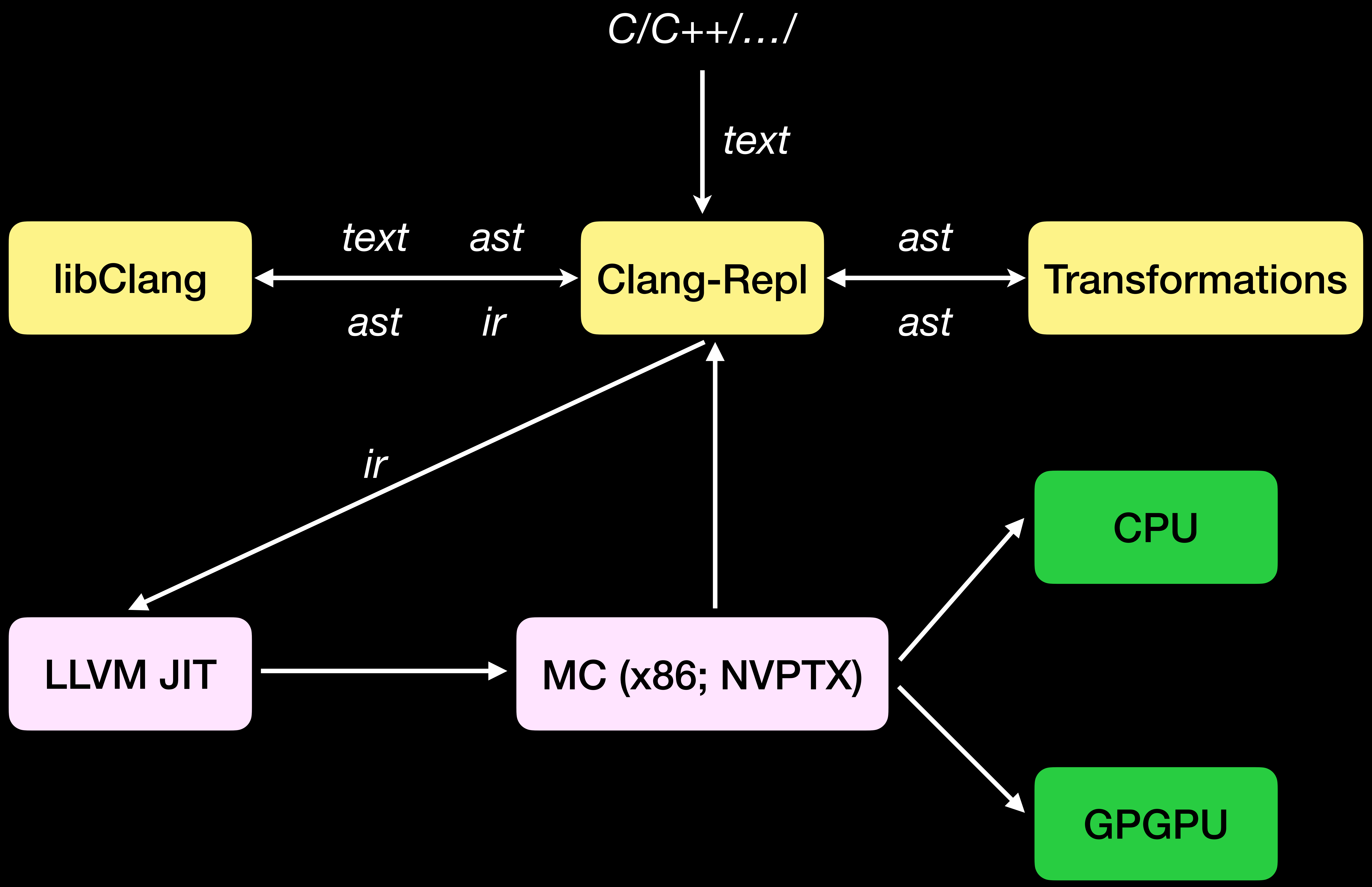
## p3-ex2.cpp: A Simple C++ REPL

```
llvm::LineEditor LE("pldi-cpp-repl");
while (std::optional<std::string> Line = LE.readLine()) {
  if (*Line == "%quit")
    break;
  if (auto Err = Interp->ParseAndExecute(*Line)) {
    …
  }
}
```

Adds a partial translation unit

# REPL

## Data Flow



# Exercise 3
# Bridging Compiled and Interpreted C++

Frontend

Interpreter

# Exercise 3

## p3-ex3.cpp: Compiler-As-A-Service (CaaS)

- Transporting results from compiled to interpreted code and back
- Frontend support for taking control of object lifetime
- Frontend support handling C++ semantics such as non-assignable types
- clang::Value
  - An efficient, ref-counted, small-buffer optimized facility to transport results
  - Supports pretty printing and type conversion operations

More Info in Jun Zhang's Presentation and the related RFC

# Exercise 3

## p3-ex3.cpp: Bridging Compiled and Interpreted C++ with With CaaS

```
// Create a value to store the transport the result from JIT.
clang::Value V;
Interp->ParseAndExecute(R"(extern "C" int sq(int x){return x*x;}
                                       sq(12)
                                      )", &V);

printf("From JIT: square(12)=%d\n", V.getInt());

// Or just get a function pointer and call it in compiled code:
auto SymAddr = ExitOnErr(Interp->getSymbolAddress("sq"));
auto sqPtr = SymAddr.toPtr<int(*)(int)>();
printf("From compiled code: square(13)=%d\n", sqPtr(13));
```

# Exercise 4
# Compiler-As-A-Service With C, Python

Frontend

Interpreter

# Exercise 4

## Programmatically Instantiating C++ Templates

- Create a library containing CaaS blocks and create an interface for it
- Build a C binary that can programmatically instantiate and call a C++ template function
- Wire this infrastructure to Python via its standard facilities

# Exercise 4

## p3-ex4-lib: A Library Capable of Instantiating a Template

Returns a declaration given a string

```
void Clang_Parse(const char* Code);

void* Clang_LookupName(...);

unsigned Clang_GetFunctionAddress(...);

void* Clang_CreateObject(...);

void* Clang_InstantiateTemplate(...);
```

Returns the address in memory of JIT'd function

Allocates storage for a declaration

# Exercise 4

## p3-ex4.c: Incorporates p3-ex4-lib in a C binary

```
const char* Code = "void* operator new(__SIZE_TYPE__, void* __p) noexcept;"
                   "\n #include <typeinfo> \n”
                   "extern \"C\" int printf(const char*,...);"
                   "class A {};"
                   "struct B {"
                   "  template<typename T>"
                   "  void callme(T) {"
                   "    printf(\" Instantiated with [%s] \\n \", typeid(T).name());"
                   "  }"
                   "};";
```

# Exercise 4

## p3-ex4.c: Incorporates p3-ex4-lib in a C binary

```c
int main(int argc, char **argv) {
  Clang_Parse(Code);
  Decl_t TemplatedClass = Clang_LookupName("B", /*Context=*/0);
  T = Clang_LookupName("A", /*Context=*/0);

  // Instantiate B::callme with the given types
  Decl_t Instantiation = Clang_InstantiateTemplate(TemplatedClass, "callme", "B");

  // Get the symbol to call
  typedef void (*fn_def)(void*);
  fn_def callme_fn_ptr = (fn_def) Clang_GetFunctionAddress(Instantiation);

  // Create objects of type A
  void* NewA = Clang_CreateObject(T);

  callme_fn_ptr(NewA);
}
```

# Exercise 4

**instantiate_cpp_template.py: Instantiating C++ Templates On-Demand With Python**

- Utilize p3-ex4-lib via ctypes
- Build a python wrapper API for the the functions in p3-ex4-lib
    - InterOpLayerWrapper — responsible for the C++ template instantiations
    - TemplateWrapper — finds and matches the C++ template arguments
    - CallCPPFunc — calls the low-level JIT’d function pointers

# Exercise 4

## instantiate_cpp_template.py

```
import ctypes

libInterop = ctypes.CDLL("p3-ex4-lib.so")
# tell ctypes which function to call and what are the expected in/out types.
_cpp_compile = libInterop.Clang_Parse
_cpp_compile.argtypes = [ctypes.c_char_p]

def cpp_compile(arg):
  return _cpp_compile(arg.encode("ascii"))
# define some classes to play with
cpp_compile(r"""\
void* operator new(__SIZE_TYPE__, void* __p) noexcept;
extern "C" int printf(const char*,...);
class A {};
struct B : public A {
  template<typename T, typename U>
  void callme(T, U) { printf(" call me may B! \n"); }
};
""")
```

# Exercise 4

## instantiate_cpp_template.py

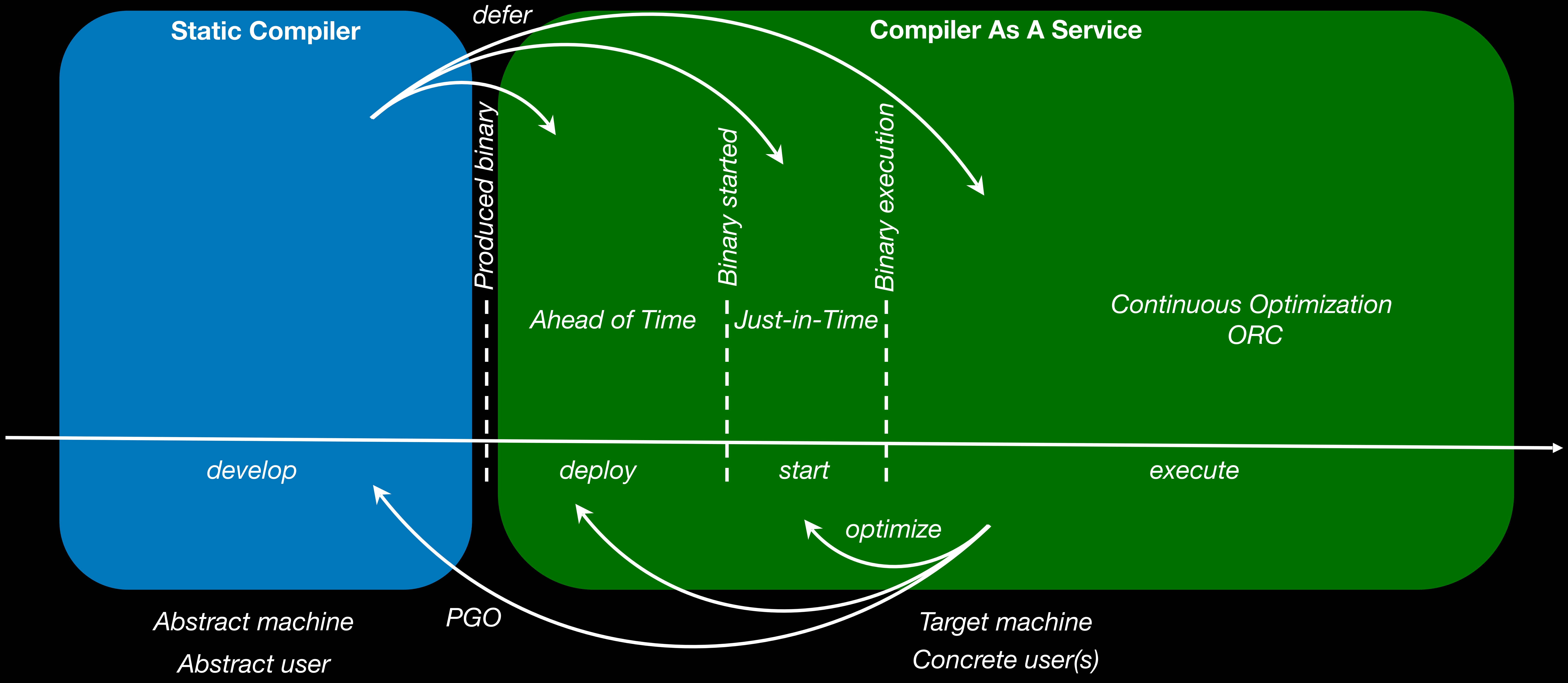
```
# Initialize our C++ interoperability layer wrapper
gIL = InterOpLayerWrapper()

if __name__ == '__main__':
  # Create a couple of types to play with
  CppA = type('A', (), {
    'handle'  : gIL.get_scope('A'),
    '__new__' : cpp_allocate
  })
  h = gIL.get_scope('B')
  CppB = type('B', (A,), {
     'handle'  : h,
     '__new__' : cpp_allocate,
     'callme'  : TemplateWrapper(h, 'callme')
  })
  # Connect to C++ classes
  a = CppA()
  b = CppB()

  # Explicit template instantiation and execution
  b.callme['A, int'](a, 42)

  # Implicit template instantiation and execution
  b.callme(a, 42)
```

# CaaS Interaction And Opportunities



# Bonus Exercise
# A Simple CUDA REPL

Frontend

Interpreter

# Bonus Exercise

## p3-ex-bonus.cpp: A Simple CUDA REPL

```
clang::IncrementalCompilerBuilder CB;
CB.SetCompilerArgs({"-std=c++20"});

// Create the device compiler.
CB.SetOffloadArch("sm_35");
auto DeviceCI = ExitOnErr(CB.CreateCudaDevice());

// Create the incremental compiler instance.
auto CI = ExitOnErr(CB.CreateCudaHost());

// Create the interpreter instance.
auto Interp =
    ExitOnErr(Interpreter::createWithCUDA(std::move(CI),
                                          std::move(DeviceCI)));
```

Creates device compiler instance

Creates host compiler instance

# Conclusions

- LLVM is widely used across industry and academia, has an active community
- LLVM may be a useful for your project
- LLVM contributions can have a wide impact
- We're always excited to welcome new people to the community
  - We can connect you with community members and mentors
  - We can connect you with Good First Issues to get started
- Come and chat to us around the conference!