Building Compilers With MLIR

An intro

Mehdi Amini - NVIDIA

Cambridge “summer” School - 9/8/2025

Agenda

- What’s a compiler?
- What’s MLIR and why?
- Core MLIR concepts
- MLIR in practice:
  - Play with the C++ API: Traverse the IR, print what we can.
- Write a pass:
  - Understand the scheduling
  - Mutate the IR
  - Debugging
  - Diagnostics and remarks

Start by cloning: https://github.com/joker-eph/playing-with-mlir

```
git clone git@github.com:joker-eph/playing-with-mlir.git
```

Pull the container:

```
docker pull jokereph/mlir-tutorial:debug
```

Large number of public talks

LLVM Dev Meeting 2020:

MLIR Tutorial [Video] [Slides]

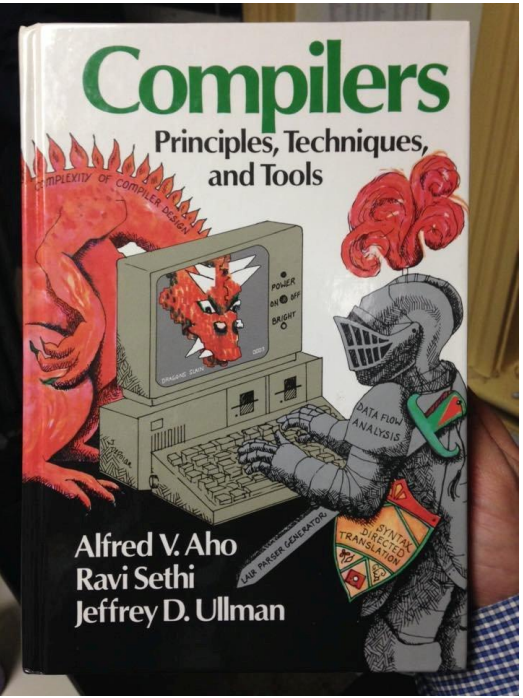
Lot more content, 89 technical talks from our public meetings:

https://mlir.llvm.org/talks/

Very active community: 25 out of 54 talks about MLIR at EuroLLVM 2023!

https://www.youtube.com/@LLVMPROJ/search?query=MLIR

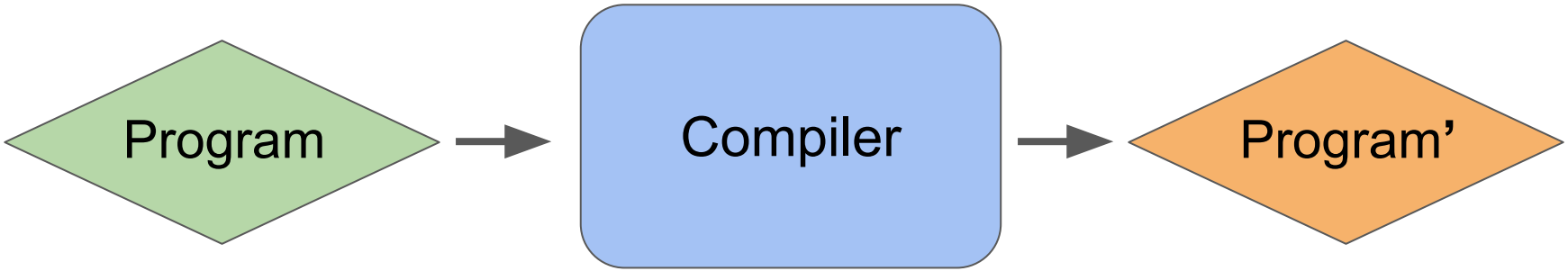
What's a compiler anyway?



- Just another software which takes some "data" in and produces "data" out.
  Turns out the "data" is itself a program, which makes the compiler "meta" somehow
- "Pure": (compilers a often written in functional languages)
  - No I/O, easy to keep stateless
  - Deterministic
  - Sequential (in general): no race bugs
  - Easy to test

=> Compilers are easy!

Why a compiler



Program’ is “similar” to the original Program, what changed?

- Program may be textual representation and Program’ executable?
  (but this is the case for an “assembler” as well, is the “assembler” a compiler?)
- Program’ may be source in another language? (Python-to-C++?)
- Program’ may be “faster”?
- Program’ may be “instrumented”?
- Program’ may be using an accelerator?
- …

What’s a program?

According to ChatGPT:

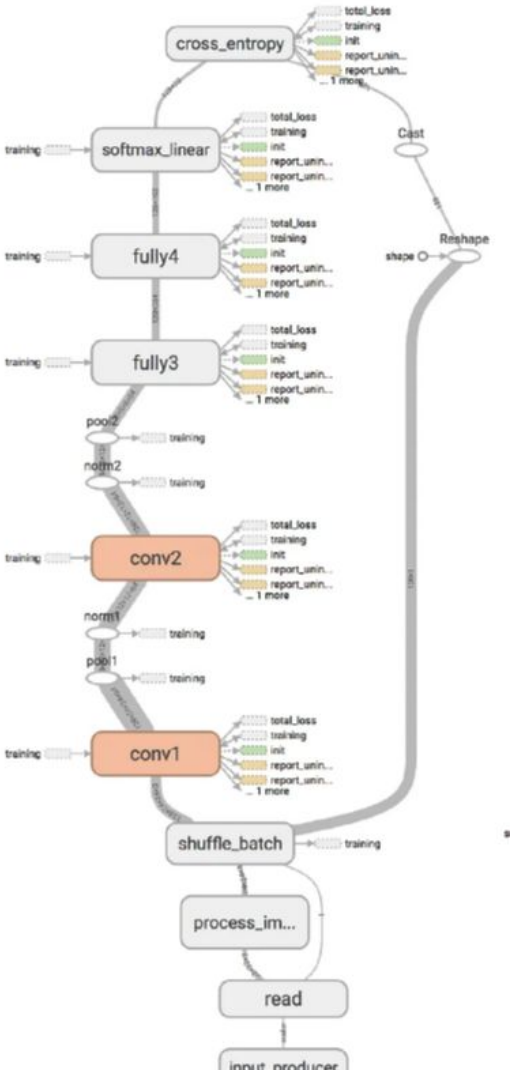
A program, in the context of computing, is a set of instructions that a computer can execute to perform a specific task or function. It consists of a ***sequence of commands or statements written in a programming language****, which the computer's processor can understand and execute.*

What's a program?

According to ChatGPT:

A program, in the context of computing, is a set of instructions that a computer can execute to perform a specific task or function. It consists of a ***sequence of commands or statements written in a programming language****, which the computer's processor can understand and execute.*

```
select
sum(total_revenue_per_order) as total_revenue
,avg(total_revenue_per_order) as average_revenue_per_order
,min(total_revenue_per_order) as smallest_order
,max(total_revenue_per_order) as largest_order
from value_per_order
```

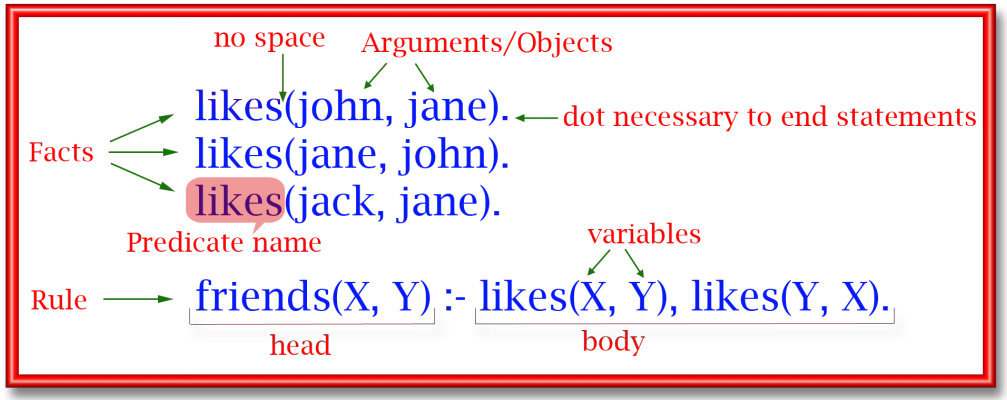


What's a program?

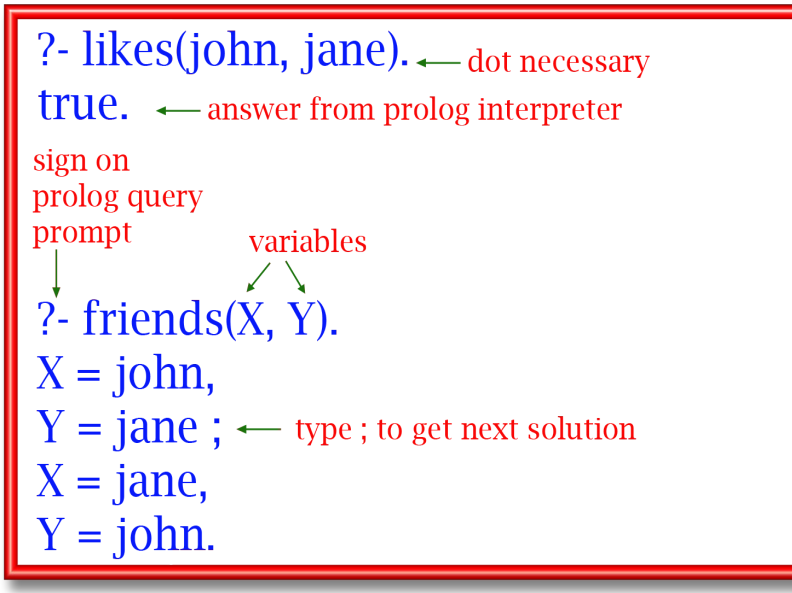
According to ChatGPT:

*A program, in the context of computing, is a set of instructions that a computer can execute to perform a specific task or function. It consists of a **sequence of commands or statements written in a programming language**, which the computer's processor can understand and execute.*

Program Window



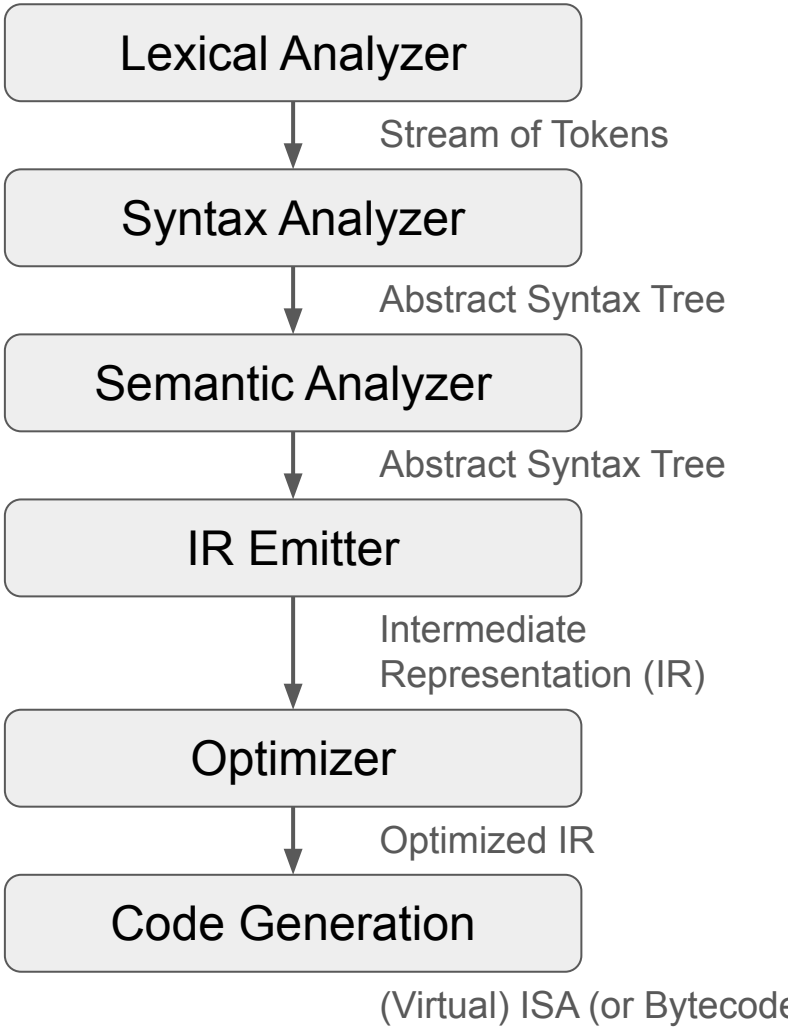
Query Window



What's a program?

According to ChatGPT:

A program, in the context of computing, is a set of instructions that a computer can execute to perform a specific task or function. It consists of a ***sequence of commands or statements written in a programming language****, which the computer's processor can understand and execute.*

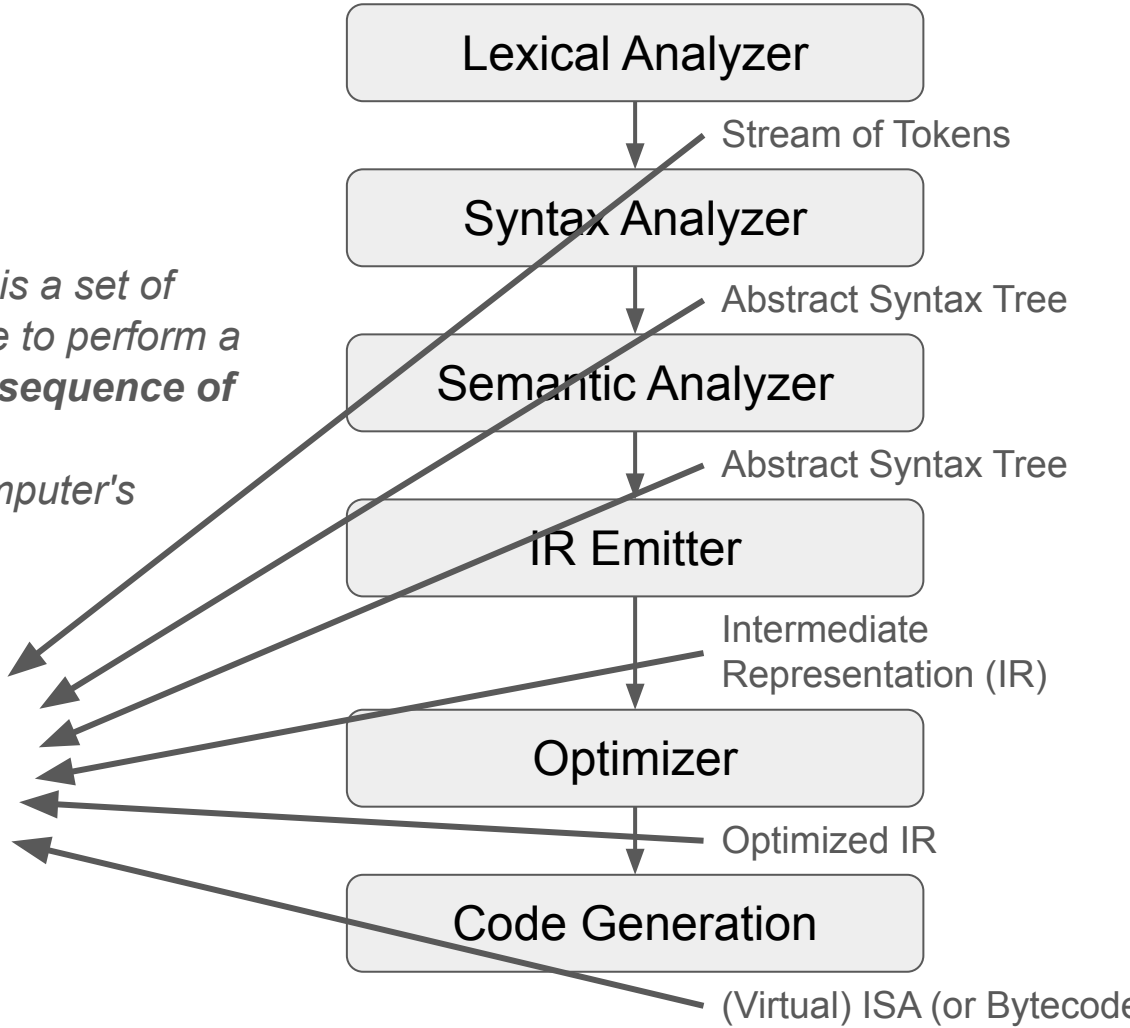


What's a program?

According to ChatGPT:

A program, in the context of computing, is a set of instructions that a computer can execute to perform a specific task or function. It consists of a ***sequence of commands or statements written in a programming language****, which the computer's processor can understand and execute.*

Different representations of the same program!

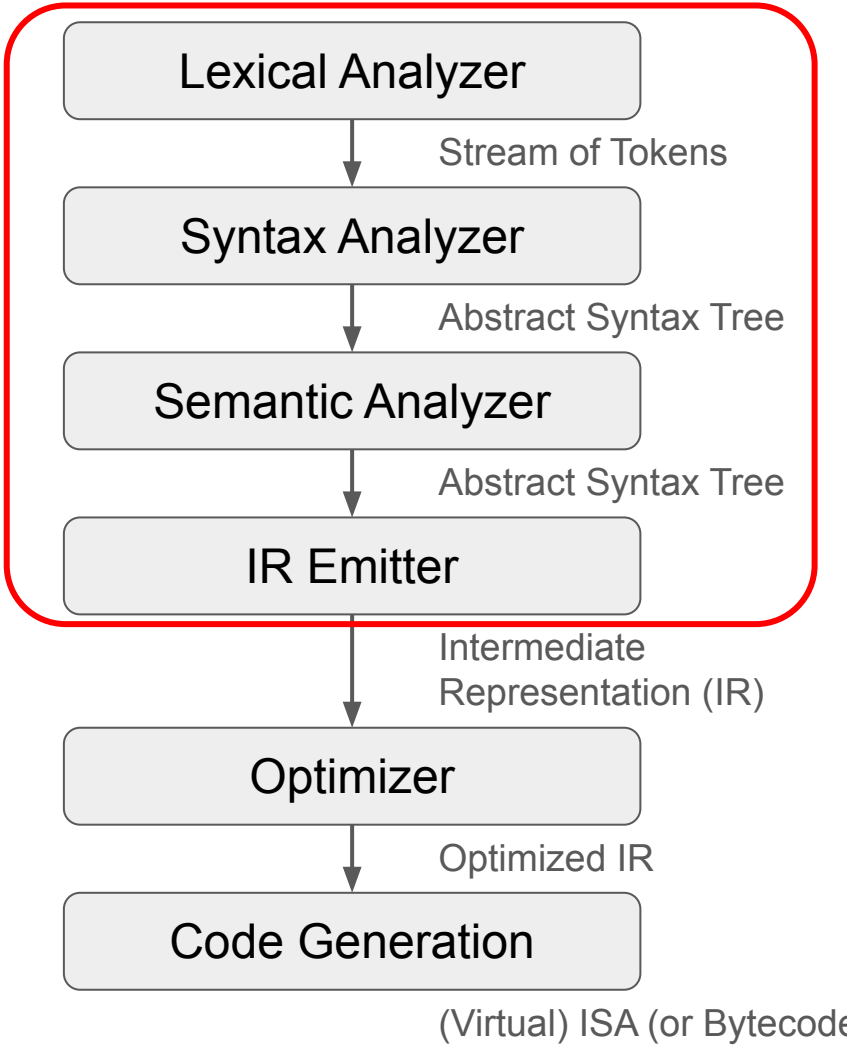


Frontend of the compiler

Main role: from text to a structured representation.

- Grammar
- Type system
- Diagnostics

Operates on “AST”: faithful structure, not easy to transform!



Frontend of the compiler

Main role: from text to a structured representation.

- Grammar
- Type system
- Diagnostics

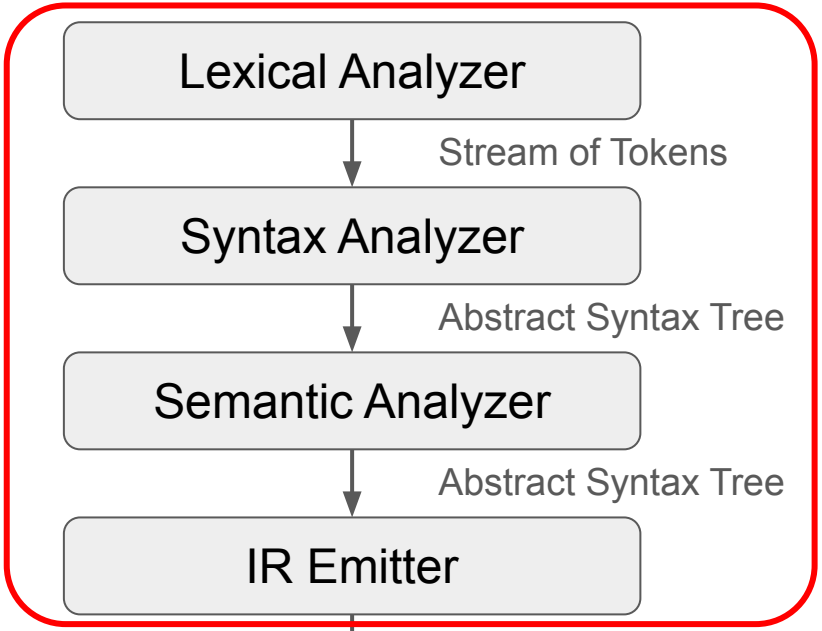
Operates on “AST”: faithful structure, not easy to transform!

clang -Xclang -ast-dump

```
1
2  int square(int num) {
3      return num * num;
4  }
5
```

```
`-FunctionDecl 0xd0c28d8 <<source>:2:1, line:4:1> line:2:5 square 'int (int)'
  |-ParmVarDecl 0xd0c2800 <col:12, col:16> col:16 used num 'int'
  `-CompoundStmt 0xd0c2a70 <col:21, line:4:1>
    `-ReturnStmt 0xd0c2a60 <line:3:5, col:18>
      `-BinaryOperator 0xd0c2a40 <col:12, col:18> 'int' '*'
        |-ImplicitCastExpr 0xd0c2a10 <col:12> 'int' <LValueToRValue>
        | `-DeclRefExpr 0xd0c29d0 <col:12> 'int' lvalue ParmVar 0xd0c2800 'num' 'int'
        `-ImplicitCastExpr 0xd0c2a28 <col:18> 'int' <LValueToRValue>
          `-DeclRefExpr 0xd0c29f0 <col:18> 'int' lvalue ParmVar 0xd0c2800 'num' 'int'
```

https://godbolt.org/z/1br84hxdK



Frontend of the compiler

Main role: from text to a structured representation.

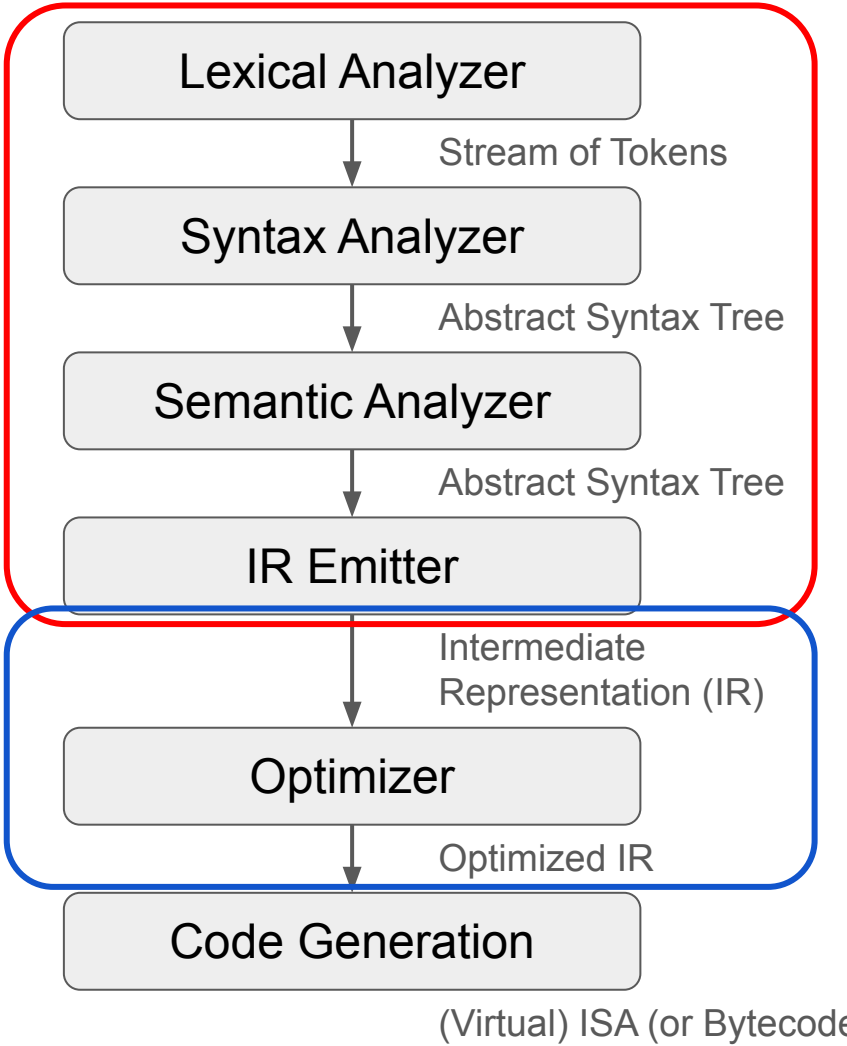
- Grammar
- Type system
- Diagnostics

Operates on “AST”: faithful structure, not easy to transform!

“Middle-end” of the compiler

Transform the program, usually to “optimize”

Operates on IR: designed for analysis and transformations, does not contains “source” information like the AST.



```
1
2  int square(int num) {
3      return num * num;
4  }
5
```

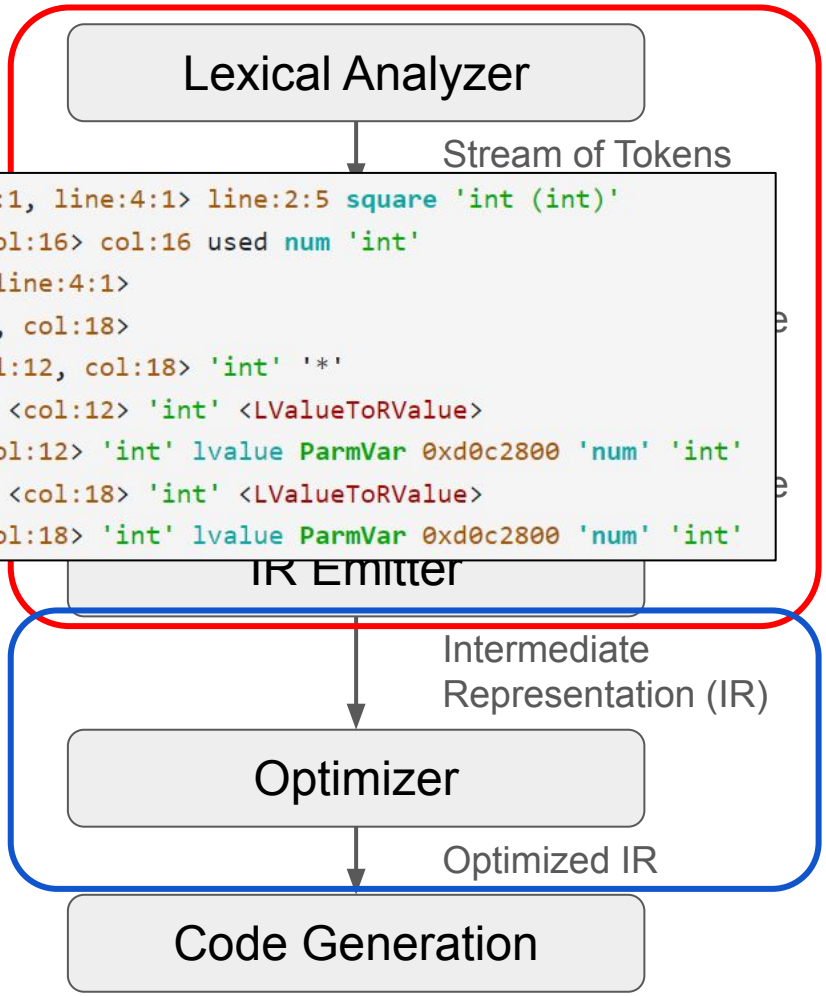
```
`-FunctionDecl 0xd0c28d8 <<source>:2:1, line:4:1> line:2:5 square 'int (int)'
  |-ParmVarDecl 0xd0c2800 <col:12, col:16> col:16 used num 'int'
  `-CompoundStmt 0xd0c2a70 <col:21, line:4:1>
    `-ReturnStmt 0xd0c2a60 <line:3:5, col:18>
      `-BinaryOperator 0xd0c2a40 <col:12, col:18> 'int' '*'
        |-ImplicitCastExpr 0xd0c2a10 <col:12> 'int' <LValueToRValue>
        | `-DeclRefExpr 0xd0c29d0 <col:12> 'int' lvalue ParmVar 0xd0c2800 'num' 'int'
        `-ImplicitCastExpr 0xd0c2a28 <col:18> 'int' <LValueToRValue>
          `-DeclRefExpr 0xd0c29f0 <col:18> 'int' lvalue ParmVar 0xd0c2800 'num' 'int'
```

“Middle-end” of t

Transform the program
Operates on IR: designed for analysis and transformations, does not contains “source” information like the AST.

clang -emit-llvm -S -o -

```
1  define dso_local noundef i32 @square(int)(i32 noundef %0) #0 {
2    %2 = alloca i32, align 4
3    store i32 %0, ptr %2, align 4
4    %3 = load i32, ptr %2, align 4
5    %4 = load i32, ptr %2, align 4
6    %5 = mul nsw i32 %3, %4
7    ret i32 %5
8  }
```



Frontend of the compiler

Main role: from text to a structured representation.

- Grammar
- Type system
- Diagnostics

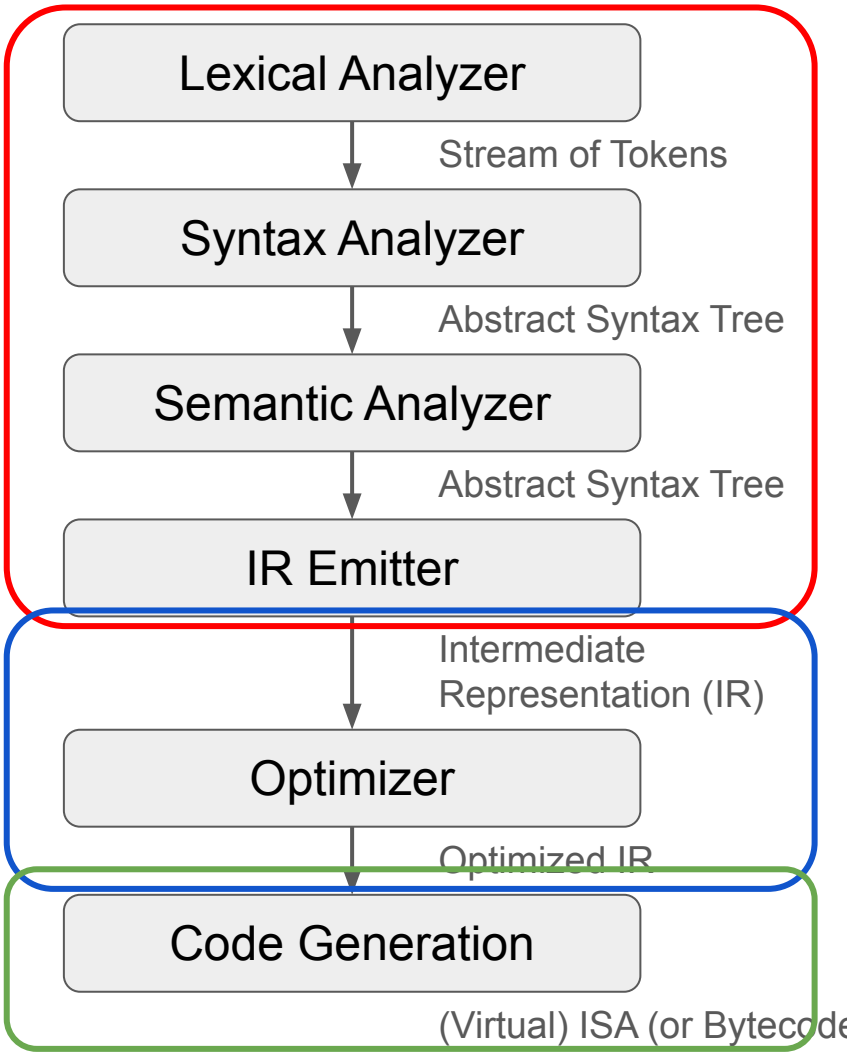
Operates on “AST”: faithful structure, not easy to transform!

“Middle-end” of the compiler

Transform the program, usually to “optimize”
Operates on IR: designed for analysis and transformations, does not contains “source” information like the AST.

“Backend” of the compiler

From IR to “ISA”: select the available instruction on the target.



Frontend of the compiler

Main role: from text to a structured representation.

- Grammar
- Type system
- Diagnostics

Operates on “AST”: faithful structure, not easy to transform!

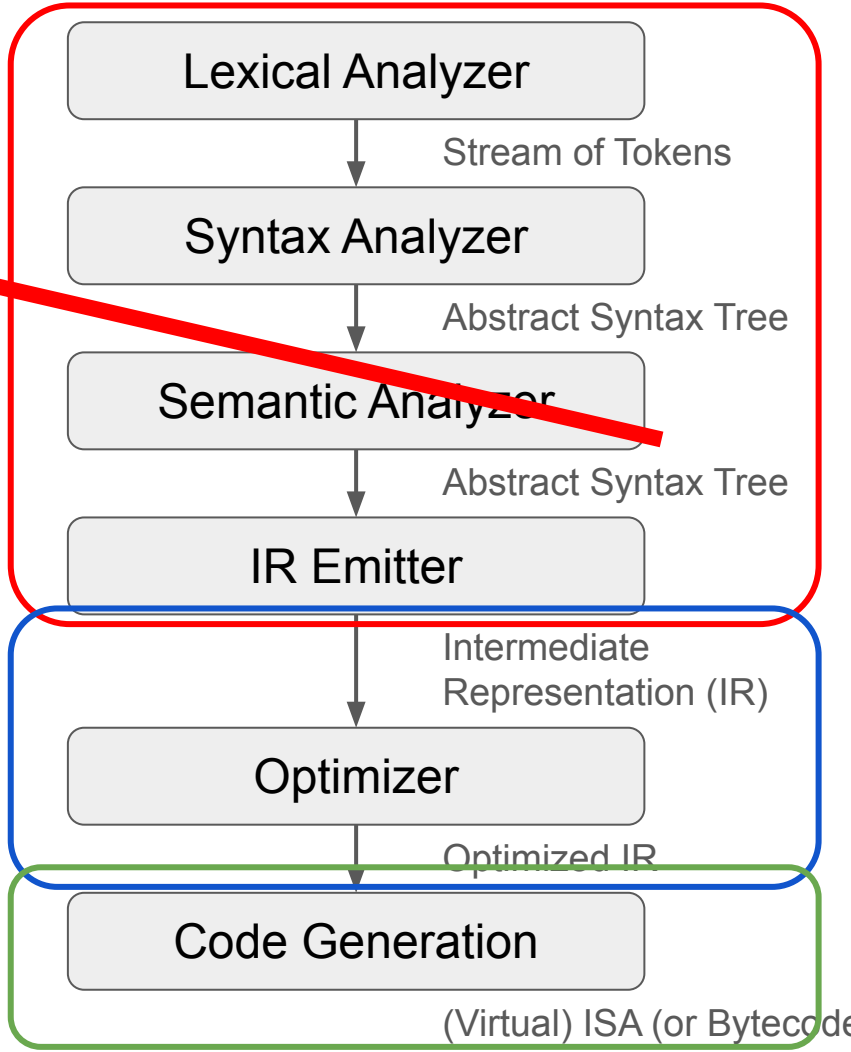
“Middle-end” of the compiler

Transform the program, usually to “optimize”
Operates on IR: designed for analysis and transformations, does not contains “source” information like the AST.

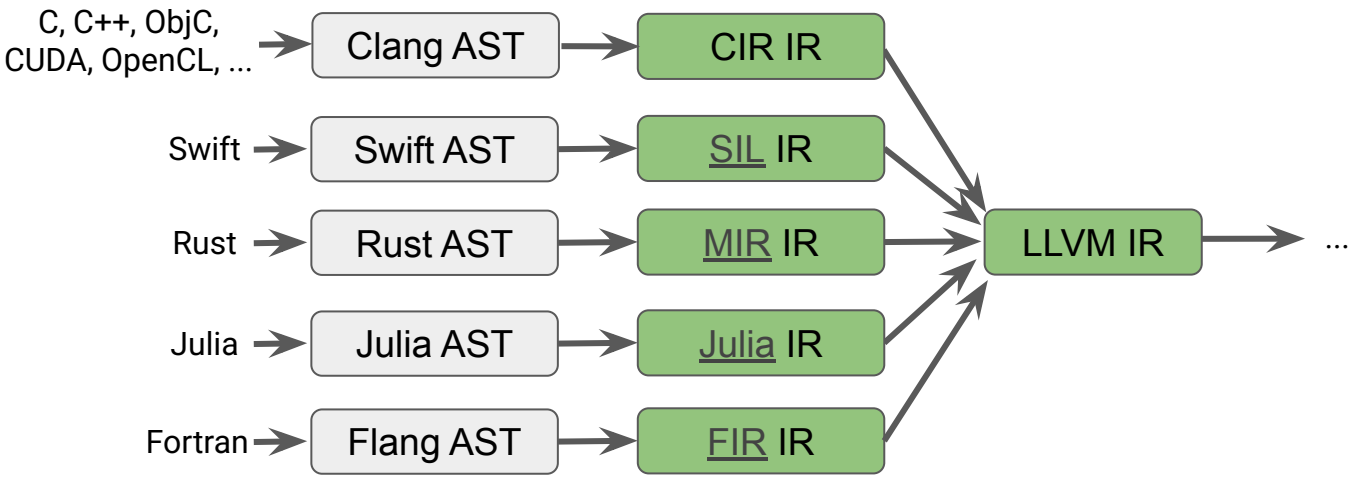
MLIR

“Backend” of the compiler

From IR to “ISA”: select the available instruction on the target.



Modern languages pervasively invest in high level IRs

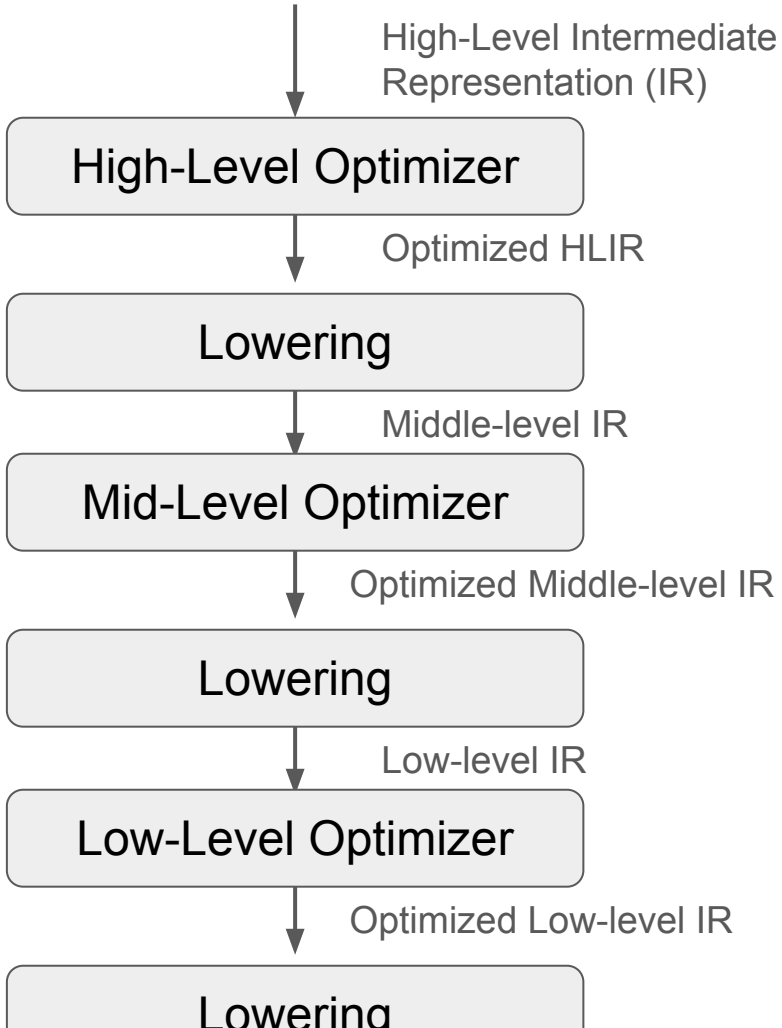


- Language specific optimizations
- Dataflow driven type checking - e.g. definitive initialization, borrow checker
- Progressive lowering from high level abstractions

Compiler Stack

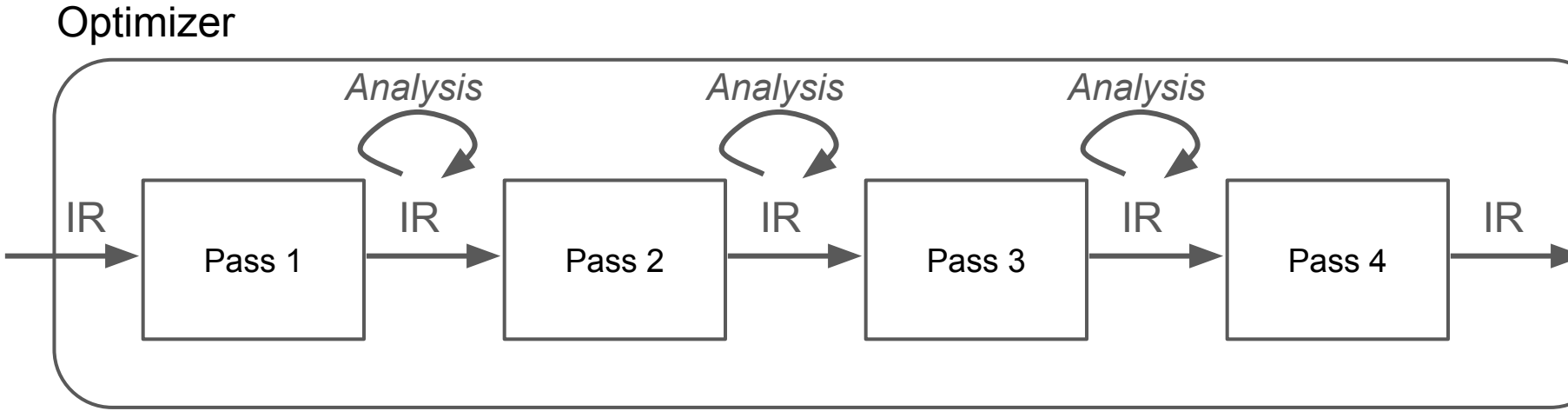
Progressive “lowering” of the abstraction level: different kind of optimizations at each level.

To some extent, each level is the “backend” for the previous level, and the “frontend” for the next level.



Compiler Passes

The program represented with the IR goes through a “pipeline” of passes (e.g. “loop unrolling”, “common subexpressions elimination”, “dead-store elimination”, “inlining”, …)



Natural granularity for testing!

Compiler Passes: clang -emit-llvm -O3 -mllvm -print-pipeline-passes

annotation2metadata,forceattrs,declare-to-assign,inferattrs,coro-early,function<eager-inv>(lower-expect,simplifycfg<bonus-inst-threshold=1;no-forward-switch-cond;no-switch-range-to-icmp;no-switch-to-lookup;keep-loops;no-hoist-common-insts;no-sink-common-insts;speculate-blocks;simplify-cond-branch>,sroa<modify-cfg>,early-cse<>,callsite-splitting),openmp-opt,ipsccp,called-value-propagation,globalopt,function<eager-inv>(mem2reg,instcombine<max-iterations=1;no-use-loop-info;no-verify-fixpoint>,simplifycfg<bonus-inst-threshold=1;no-forward-switch-cond;switch-range-to-icmp;no-switch-to-lookup;keep-loops;no-hoist-common-insts;no-sink-common-insts;speculate-blocks;simplify-cond-branch>),always-inline,require<globals-aa>,function(invalidate<aa>),require<profile-summary>,cgscc(devirt<4>(inline,function-attrs<skip-non-recursive-function-attrs>,argpromotion,openmp-opt-cgscc,function<eager-inv;no-rerun>(sroa<modify-cfg>,early-cse<memssa>,speculative-execution<only-if-divergent-target>,jump-threading,correlated-propagation,simplifycfg<bonus-inst-threshold=1;no-forward-switch-cond;switch-range-to-icmp;no-switch-to-lookup;keep-loops;no-hoist-common-insts;no-sink-common-insts;speculate-blocks;simplify-cond-branch>,instcombine<max-iterations=1;no-use-loop-info;no-verify-fixpoint>,aggressive-instcombine,libcalls-shrinkwrap,tailcallelim,simplifycfg<bonus-inst-threshold=1;no-forward-switch-cond;switch-range-to-icmp;no-switch-to-lookup;keep-loops;no-hoist-common-insts;no-sink-common-insts;speculate-blocks;simplify-cond-branch>,reassociate,constraint-elimination,loop-mssa(loop-instsimplify,loop-simplifycfg,licm<no-allowspeculation>,loop-rotate<header-duplication;no-prepare-for-lto>,licm<allowspeculation>,simple-loop-unswitch<nontrivial;trivial>),simplifycfg<bonus-inst-threshold=1;no-forward-switch-cond;switch-range-to-icmp;no-switch-to-lookup;keep-loops;no-hoist-common-insts;no-sink-common-insts;speculate-blocks;simplify-cond-branch>,instcombine<max-iterations=1;no-use-loop-info;no-verify-fixpoint>,loop(loop-idiom,indvars,loop-deletion,loop-unroll-full),sroa<modify-cfg>,vector-combine,mldst-motion<no-split-footer-bb>,gvn<>,sccp,bdce,instcombine<max-iterations=1;no-use-loop-info;no-verify-fixpoint>,jump-threading,correlated-propagation,adce,memcpyopt,dse,move-auto-init,loop-mssa(licm<allowspeculation>),coro-elide,simplifycfg<bonus-inst-threshold=1;no-forward-switch-cond;switch-range-to-icmp;no-switch-to-lookup;keep-loops;hoist-common-insts;sink-common-insts;speculate-blocks;simplify-cond-branch>,instcombine<max-iterations=1;no-use-loop-info;no-verify-fixpoint>),function-attrs,function(require<should-not-run-function-passes>),coro-split)),deadargelim,coro-cleanup,globalopt,glo

OK, Compiler looks cool, but what's MLIR?

Not a compiler, and definitely nothing to do with "ML".

- Framework to build a compiler IR: define your type system, operations, etc.
- Toolbox covering your compiler infrastructure needs
  - Diagnostics, pass-management infrastructure, multi-threading, testing tools, etc.
- Batteries-included:
  - Various code-generation components / lowering strategies
  - Tooling for accelerator support (GPUs)
- Allow different levels of abstraction to freely co-exist
  - Abstractions can better target specific areas with less high-level information lost
  - Progressive lowering simplifies and enhances transformation pipelines
  - No arbitrary boundary of abstraction, e.g. host and device code in the same IR at the same time
- LLVM-inspired

Underlying MLIR development

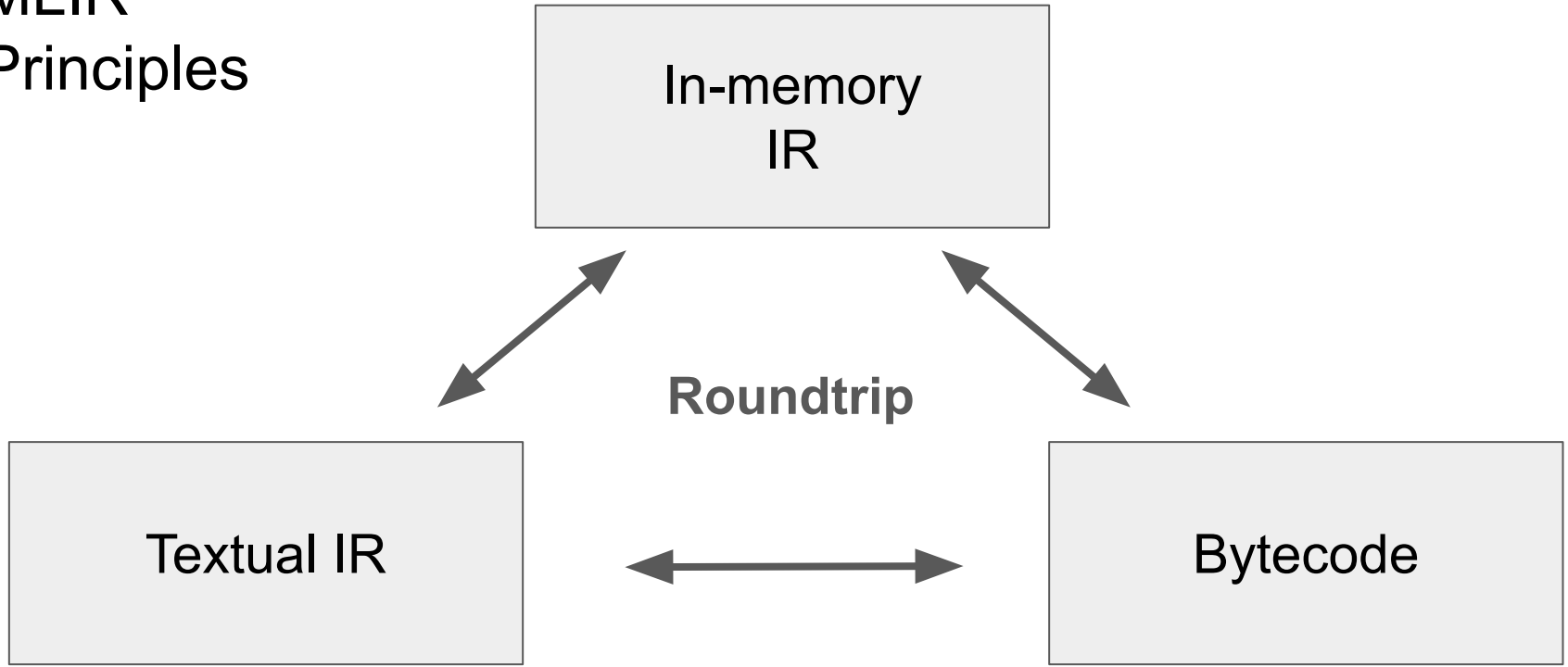
We “sell” a combination of:

- **Time to market**: minimize the time “from idea to prototype”, easier transition for “research to production”
- Allow you to **focus on your value-add** (do you want to reimplement everything?)
- **Flexibility**: encourage modular design (dialect), allow you to adjust and re-iterate, resilient to initial mistakes.
- **Built for production** from day-1: heavily optimized behind the scene.
  -> minimize the effort to go “from prototype to a product”
  Inspired by LLVM, fixing all the mistakes!
- **Modular**: “you pay for what you use” footprint
  - CoreIR & builtin dialect & printer/parser & bytecode: 2MB
  - + the pass infrastructure, instrumentation, etc.: 3MB (+1MB)
  - + canonicalize, pattern/rewrite infra, PDL compiler/interpreter: 4.8MB (+1.8MB)
- **Standardize compiler infrastructure:** don’t learn over and over the peculiarities of each compiler infrastructure. Learn MLIR and you can more easily learn about any MLIR-based compiler!
  (Translates into: good skill to have on your resume for engineer, easier hiring for companies)

Core MLIR Concepts

MLIR Principles

- Easy to manipulate, mapped to convenient APIs
- IR Invariants constraints auto-verified



- Testing
- Debugging
  (call *dump()* on any MLIR entity)

- Deployment
- Versioning: forward/backward compat

IR Core Concepts

Definition of SSA

The simplest, least constrained, definition of SSA can be given using the following informal prose:

> *A program is defined to be in SSA form if each variable is a target of exactly one assignment statement in the program text.*

https://pfalcon.github.io/ssabook/latest/book.pdf

IR Core Concepts

Very few core-defined aspects, MLIR is generic and favor extensibility:

- **Region**, either:
  - a list of basic blocks chained through their terminators to form a CF
  - a single block containing operations forming a generic directed graph (including cyclic).
- **Block**: a sequential list of Operations. *They take arguments instead of using phi nodes.*
- **Operation**: a generic single unit of “code”.
  - takes individual Values as operands,
  - produces one or more SSA Values as results.
  - A terminator operation also has a list of successors blocks, as well as arguments matching the blocks.

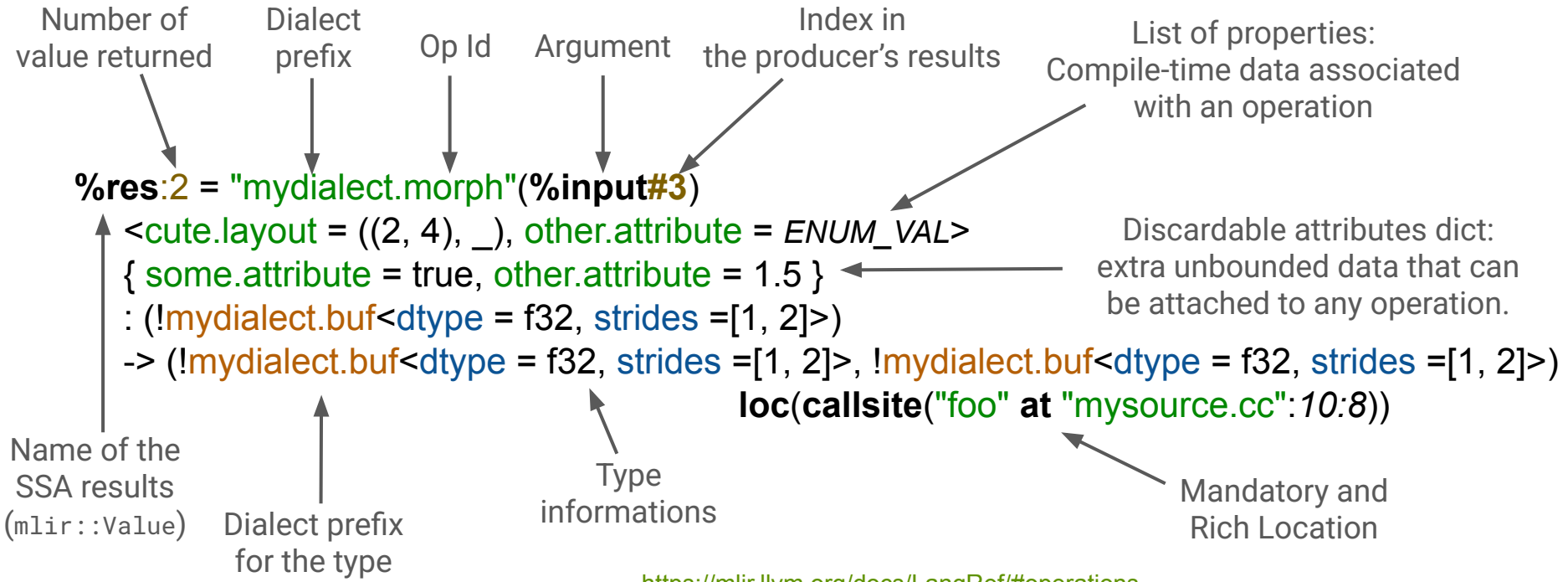
There aren’t any hard-coded structures or specific operations in MLIR:

even Module and Function are defined just as regular operations!

Wide range of applications: *Dataflow Graphs, ML Compilers, HW Design, Parallel Optimization, Heterogeneous runtime, Functional programming, Fortran Compiler, Heterogeneous code generation, Encryption, Program analysis, Zero knowledge proof compilation, Quantum, Blockchain, …*

Operations, Not Instructions

- No predefined set of instructions
- Operations are like “opaque functions” to MLIR



https://mlir.llvm.org/docs/LangRef/#operations
https://github.com/llvm/llvm-project/blob/main/mlir/include/mlir/IR/Operation.h#L31-L84

Hands-on: ir-traversal

Let’s work through the second hands on.

Recursive nesting: Operations -> Regions -> Blocks

```
%results:2 = "d.operation"() ({
  // Regions belong to Ops and can have multiple blocks.
  ^block(%argument: !d.type):
    %value = "nested.operation"() ({
      // Ops can contain nested regions.
      "d.op"() : () -> ()
    }) : () -> (!d.other_type)
    "consume.value"(%value) : (!d.other_type) -> ()
  ^other_block:
    "d.terminator"() [^block(%arg0 : !d.type)] : () ->
()
}) : () -> (!d.type, !d.other_type)
```

Region

Block

Region

- Regions are list of basic blocks nested inside of an operation.
  - Basic blocks are a list of operations: **the IR structure is recursively nested**!
- Conceptually similar to function call, but can reference SSA values defined outside.
- SSA values defined inside don't escape.

https://mlir.llvm.org/docs/Tutorials/UnderstandingTheIRStructure/
https://mlir.llvm.org/docs/LangRef/#high-level-structure

Regions

```
func.func @hello(%arg0 : i32, %arg1 : i1) {
  …
  scf.if (%arg1) {
    …
  } else {
    …
  }
  …
}
```

"CFG Regions": sequential list of operations, the control flows from one operation to the next in order (similar to LLVM)

"Graph Regions": single block with **unordered list** of operations and no terminator requirement.
Cycles in SSA def-use are also allowed!

```
my.graph @hello(%arg0 : i32, %arg1 : i1) {
  …
  B_user(%b) : …
  …
  %b = B_producer(%c) : …
  …
  %c = C_producer(%b) : …
}
```

Graph Regions

-> ML Dataflow Graphs
-> "Circuit level" logic
-> Synchronous domain
…

Future Extension to Region: Multiple-Entry, Multiple-Exit MLIR Regions
EuroLLVM 2023: slides and recording

Hands-on: ir-traversal-with-regions

Dialects: Defining Rules and Semantics for the IR

A MLIR dialect is a logical grouping including:

- A prefix (“namespace” reservation)
- A list of custom types, each its C++ class.
- A list of operations, each its name and C++ class implementation:
  - Verifier for operation invariants (e.g. *toy.print* must have a single operand)
  - Semantics (has-no-side-effects, constant-folding, CSE-allowed, ….)
- Passes: analysis, transformations, and dialect conversions.
- Possibly custom parser and assembly printer

Dialects co-exist: a program is often represent by a mix of dialects!

https://mlir.llvm.org/docs/LangRef/#dialects
https://github.com/llvm/llvm-project/blob/master/mlir/include/mlir/IR/Dialect.h#L37
https://mlir.llvm.org/docs/Tutorials/CreatingADialect/

LLVM as a dialect

```
%13 = llvm.alloca %arg0 x !llvm.double : (!llvm.i32) -> !llvm.ptr<double>
%14 = llvm.getelementptr %13[%arg0, %arg0]
          : (!llvm.ptr<double>, !llvm.i32, !llvm.i32) -> !llvm.ptr<double>
%15 = llvm.load %14 : !llvm.ptr<double>
llvm.store %15, %13 : !llvm.ptr<double>
%16 = llvm.bitcast %13 : !llvm.ptr<double> to !llvm.ptr<i64>
%17 = llvm.call @foo(%arg0) : (!llvm.i32) -> !llvm.struct<(i32, double, i32)>
%18 = llvm.extractvalue %17[0] : !llvm.struct<(i32, double, i32)>
%19 = llvm.insertvalue %18, %17[2] : !llvm.struct<(i32, double, i32)>
%20 = llvm.constant(@foo : (!llvm.i32) -> !llvm.struct<(i32, double, i32)>) :
        !llvm.ptr<func<struct<i32, double, i32> (i32)>>
%21 = llvm.call %20(%arg0) : (!llvm.i32) -> !llvm.struct<(i32, double, i32)>
```

More intro to MLIR: https://mlir.llvm.org/docs/Tutorials/Toy/

I love it: how do I start to write my dialect?

So you want to design an IR: it's a bit of an art, but not very different from other API design.

Start with the type system:

- what are the objects you want to model?
- Are you manipulating "arrays"? Scalars?
- Are they mutable?

Then the operations:

- Very dependent on the type modeling: if mutable, think about side-effects.
  dot(%a, %b, %c) vs %c = *dot(%a, %b)*
- Control-flow: imperative programing model or dataflow graph?

https://mlir.llvm.org/docs/DefiningDialects/#defining-a-dialect

ODS: Operation Definition Specification

```
def Arith_MulSIExtendedOp : Arith_Op<"mulsi_extended",
  [Pure, Commutative, AllTypesMatch<["lhs", "rhs", "low", "high"]>]> {
  let summary = "extended signed integer multiplication operation";
  let description = [{
    Performs (2*N)-bit multiplication on sign-extended operands. Returns two
    N-bit results: ...
  }];

  let arguments = (ins SignlessIntegerOrIndexLike:$lhs, SignlessIntegerOrIndexLike:$rhs);
  let results = (outs SignlessIntegerOrIndexLike:$low, SignlessIntegerOrIndexLike:$high);

  let assemblyFormat = "$lhs `,` $rhs attr-dict `:` type($lhs)";

  let hasFolder = 1;
  let hasCanonicalizer = 1;

  let extraClassDeclaration = [{
    std::optional<SmallVector<int64_t, 4>> getShapeForUnroll();
  }];
}
```

https://mlir.llvm.org/docs/DefiningDialects/Operations/
https://mlir.llvm.org/docs/Dialects/ArithOps/#arithmulsi_extended-arithmulsiextendedop

ODS: Operation Definition Specification

```
def Arith_MulSIExtendedOp : Arith_Op<"mulsi_extended",
  [Pure, Commutative, AllTypesMatch<["lhs", "rhs", "low", "high"]>]> {
  let summary = "extended signed integer multiplication operation";
  let description = [{
    Performs (2*N)-bit multiplication on sign-extended operands. Returns two
    N-bit results: ...
  }];

  let arguments = (ins SignlessIntegerOrIndexLike:$lhs, SignlessIntegerOrIndexLike:$rhs);
  let results = (outs SignlessIntegerOrIndexLike:$low, SignlessIntegerOrIndexLike:$high);

  let assemblyFormat = "$lhs `,` $rhs attr-dict `:` type($lhs)";

  %low, %high = arith.mulsi_extended %cstA, %cstB: vector<3xi8>

  let extraClassDeclaration = [{
    std::optional<SmallVector<int64_t, 4>> getShapeForUnroll();
  }];
}
```

https://mlir.llvm.org/docs/DefiningDialects/Operations/
https://mlir.llvm.org/docs/Dialects/ArithOps/#arithmulsi_extended-arithmulsiextendedop

OpenAI

TRITON DIALECT

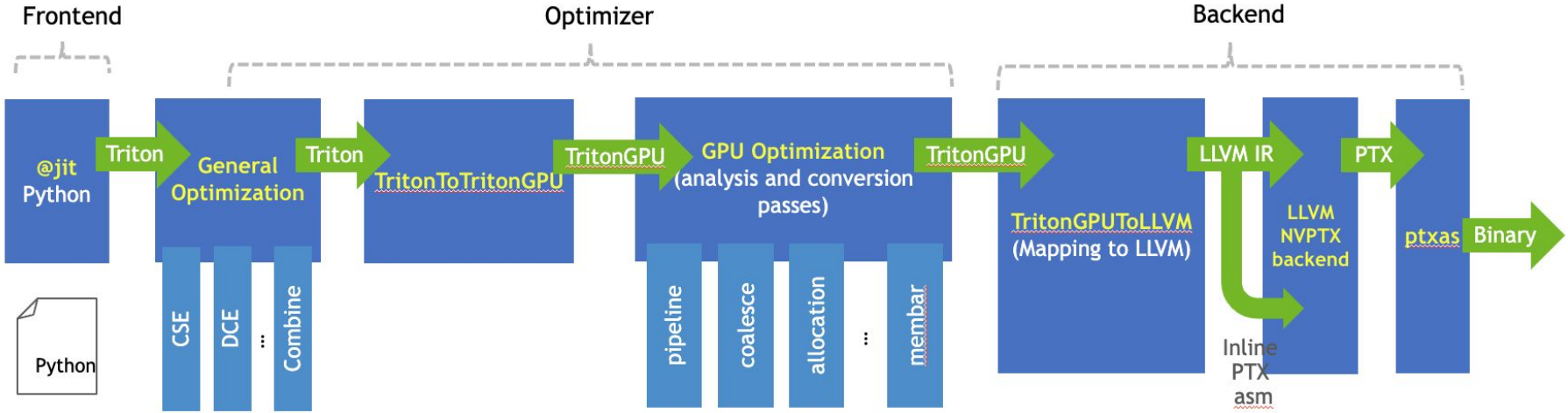
- Representing the numpy-like computation
  - load, store
  - view, splat, broadcast
  - gep, expand_dims
  - dot,reduce …
- Mixed with some std dialects
  - Arith
  - Scf

```
func @vecadd(%arg0: !tt.ptr<f32> {tt.divisibility = 4 : i32},
             %arg1: !tt.ptr<f32> {tt.divisibility = 4 : i32},
             %arg2: !tt.ptr<f32> {tt.divisibility = 4 : i32}, %arg3: i32) {
  %c256_i32 = arith.constant 256 : i32
  %0 = tt.get_program_id {axis = 0 : i32} : i32
  %1 = arith.muli %0, %c256_i32 : i32
  %2 = tt.make_range {end = 256 : i32, start = 0 : i32} : tensor<256xi32>
  %3 = tt.splat %1 : (i32) -> tensor<256xi32>
  %4 = arith.addi %3, %2 : tensor<256xi32>
  %5 = tt.splat %arg0 : (!tt.ptr<f32>) -> tensor<256x!tt.ptr<f32>>
  %6 = tt.getelementptr %5, %4 : tensor<256x!tt.ptr<f32>>
  %7 = tt.splat %arg1 : (!tt.ptr<f32>) -> tensor<256x!tt.ptr<f32>>
  %8 = tt.getelementptr %7, %4 : tensor<256x!tt.ptr<f32>>
  %9 = tt.load %6 : tensor<256xf32>
  %10 = tt.load %8 : tensor<256xf32>
  %11 = arith.addf %9, %10 : tensor<256xf32>
  %12 = tt.splat %arg2 : (!tt.ptr<f32>) -> tensor<256x!tt.ptr<f32>>
  %13 = tt.getelementptr %12, %4 : tensor<256x!tt.ptr<f32>>
  tt.store %13, %11 : tensor<256xf32>
  return
}
```

ARCHITECTURE

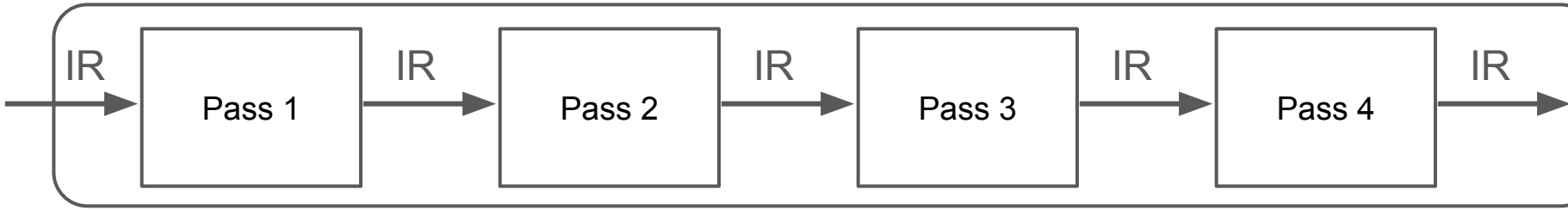
Triton-MLIR

- The lowering path is mainly made of three levels of IR
  - Triton Dialect
  - TritonGPU Dialect
    - GPU specific operations: async_wait, convert_layout etc.
    - Explicit “layout” on the IR, to represent the mapping from computation to hardware resources
  - LLVM IR with heavy use of inline PTX asm
  - Other MLIR existing dialects also being reused(Arith/Math/Func/SCF/GPUDialect)



I have a dialect now, how do I test it?

The program represented with the IR goes through a “pipeline” of passes (e.g. “loop unrolling”, “common subexpressions elimination”, “dead-store elimination”, “inlining”, …)



Conventions in MLIR/LLVM:

- `*-opt` command line tool: https://mlir.llvm.org/docs/Tutorials/MlirOpt/
- FileCheck” test: https://github.com/llvm/llvm-project/blob/main/mlir/test/Transforms/canonicalize.mlir#L1
- IR is “verified” for invariant between each pass: a pass assumes valid IR input and produces valid IR output.

Hands-on: let's play with a pass

1) Play with the scheduling
2) Debugging/logging API
3) Let's do IR mutation
4) Diagnostics and remarks

C++ entities & ownership model

MLIR has a somehow unusual model:

- The MLIRContext (like in LLVM) holds some immutable entities for the duration of its existence: Attributes and Types
  (as such: the MLIRContext must always outlive your IR)
- Many classes are actually thin wrapper around a pointer:
  - Concrete Ops, Types, and Attributes
  - Ops add "convenient" / "type safe" APIs on top of the raw pointer they encapsulate.
  - Passed by value everywhere: it's just a reference-class, always the size of a pointer.
  - Ownership of operations is their parent
  - Top-level operation (or orphan ones) should be owned in RAII class (*mlir::OwningOpRef)*

In the box...

- IR (including serialization/deserialization to text and bytecode)
- Pass Manager
- Dialect Conversion Framework
- PDL/PDLL: bytecode for dynamic loading of pattern/matching logic.
- Transform Dialect: schedule and control transformation (underlying infrastructure for a Halide-like tool)
- DataFlow engine (forward, backward, bi-directional): EuroLLVM 2023 slides and recording
- IR Reduction tool
- Tracing / Debugging infra (“compiler fuel”, bisection, etc.) doc, slides and recording
- Python Bindings (documentation)
- EmitC: generating source code (C, C++, Cuda, OpenCL, …) (documentation)…
- …

“Batteries included”:

- Tensor Abstractions
- Hyper-rectangular code generation
- Affine/Polyhedral code generation
- Presburger arithmetic library
- Sparse tensors (TACO)
- GPU code generation infrastructure support
- Multi-dimensional Vector abstractions.
- OpenMP/OpenACC
- …

Targeting:

- Fast prototypes / experiments
- Research
- Teaching
- Production compilers

Resources

- Online doc and tutorials: https://mlir.llvm.org/
- GitHub: https://github.com/llvm/llvm-project/tree/main/mlir
- Discord (~*Slack*) https://discord.gg/upPwxtwA
- Discourse Forum: https://discourse.llvm.org/c/mlir/31
- Previous tech talks: https://mlir.llvm.org/talks/
- LLVM Conferences: https://www.youtube.com/@LLVMPROJ/search?query=MLIR