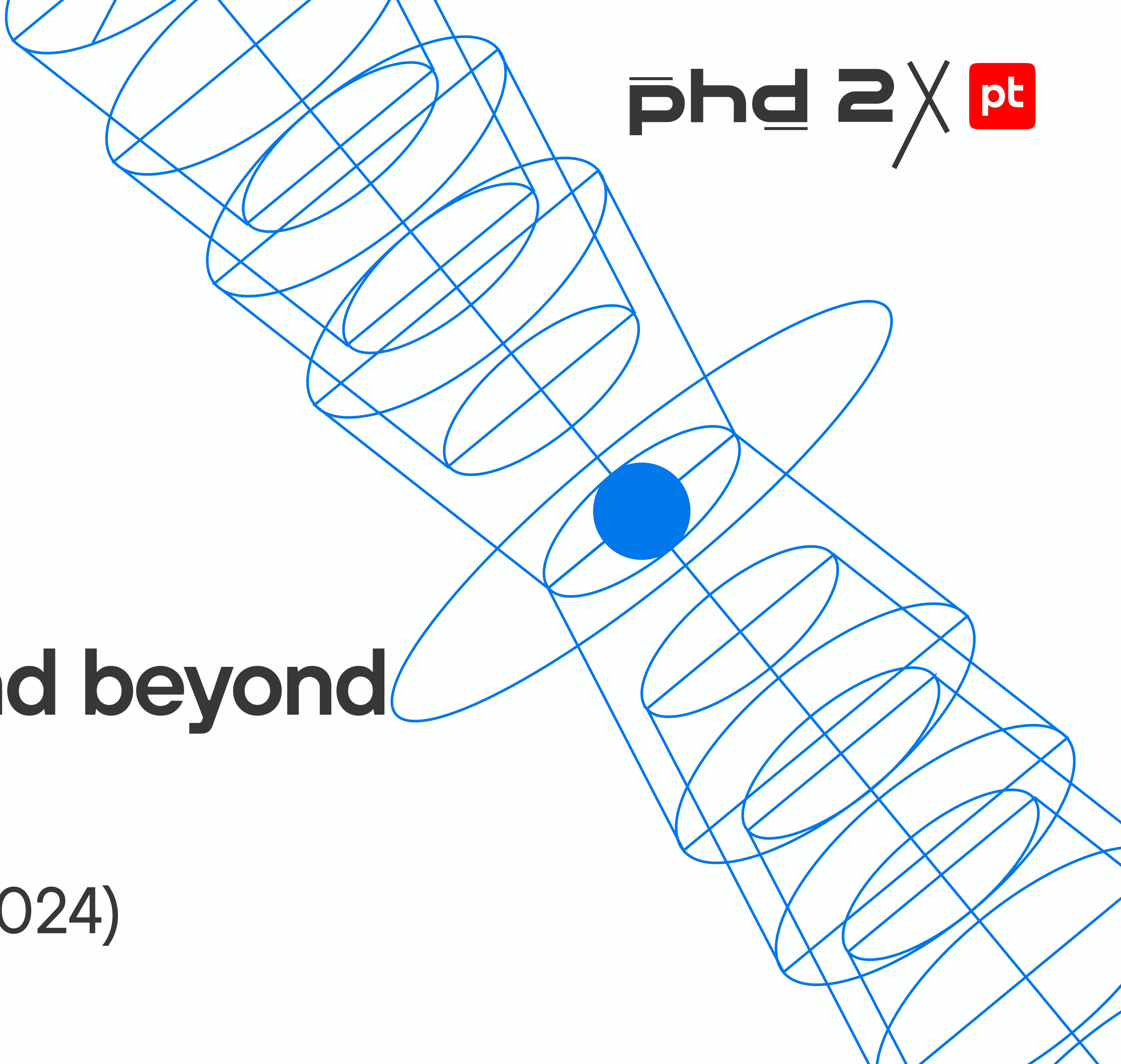
Compilers without Reinventing a Wheel:

What’s up with MLIR, Mojo 🔥 and beyond

Vasily Ryabov

Expert at Huawei (May 2024)



~~What is~~ Who is the «Novice»?

Vasily Ryabov

Since 2004 used C++, pure C, Python (2008+), mixed Python/C.

In compilers field –

last 1,5 years, from scratch.

What is a compiler?

Frontend (lexer/tokenizer + syntax)

Source code => tokens (lexemes) => AST (Abstract Syntax Tree)

Parser generators: ANTLR, PEG, …

Middle end (semantics)

Machine independent IRs (intermediate representations)

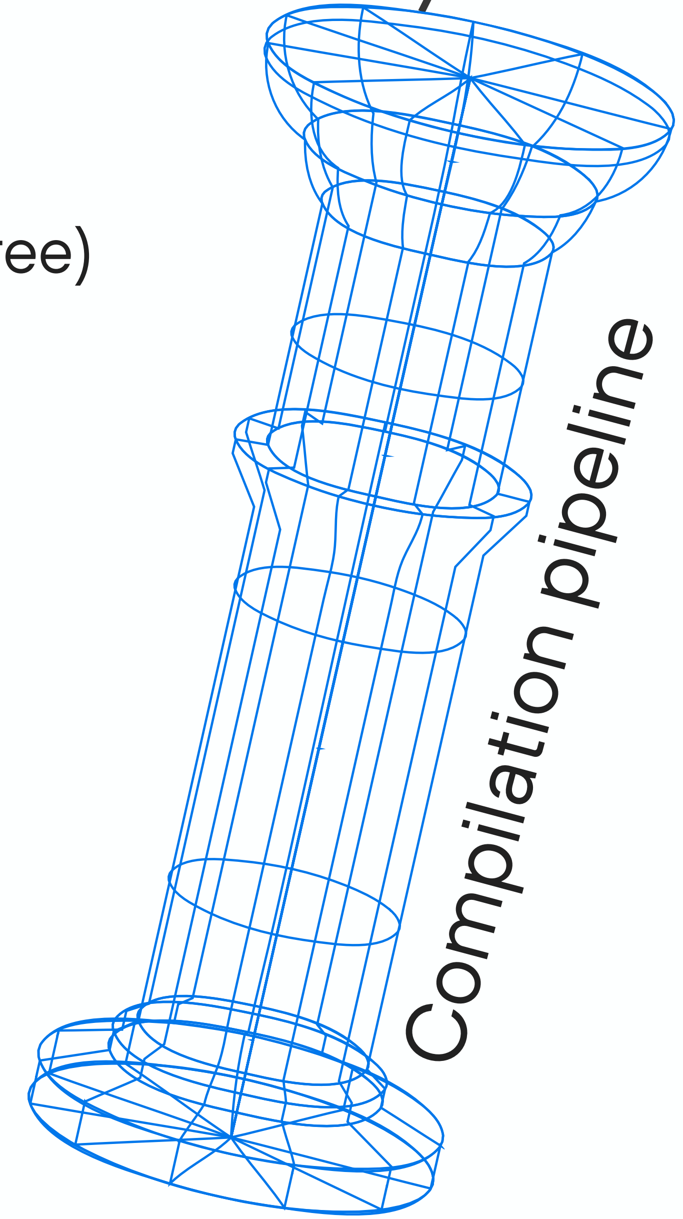
Type inference and type checks, common optimizations, exceptions, …

Backend («90% of the whole job»)

Machine dependent IRs

optimizations => register allocation => Assembler => machine code.

+ JIT (Just-in-Time) engine



What would we do?

Frontend

We will call it from Python 3.9

- in 3.10+ function `PyParser_ASTFromFile(...)` is hidden from C API

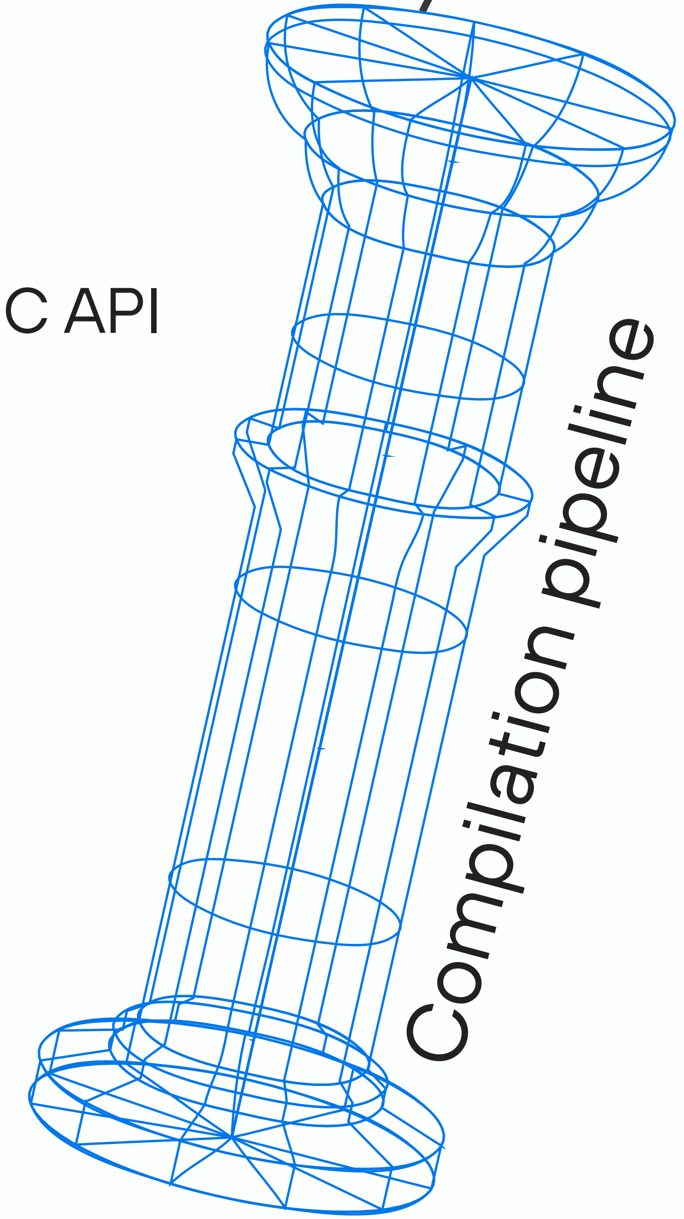
Middle end

We will generate IR with help of MLIR framework

We will lower it and pass further

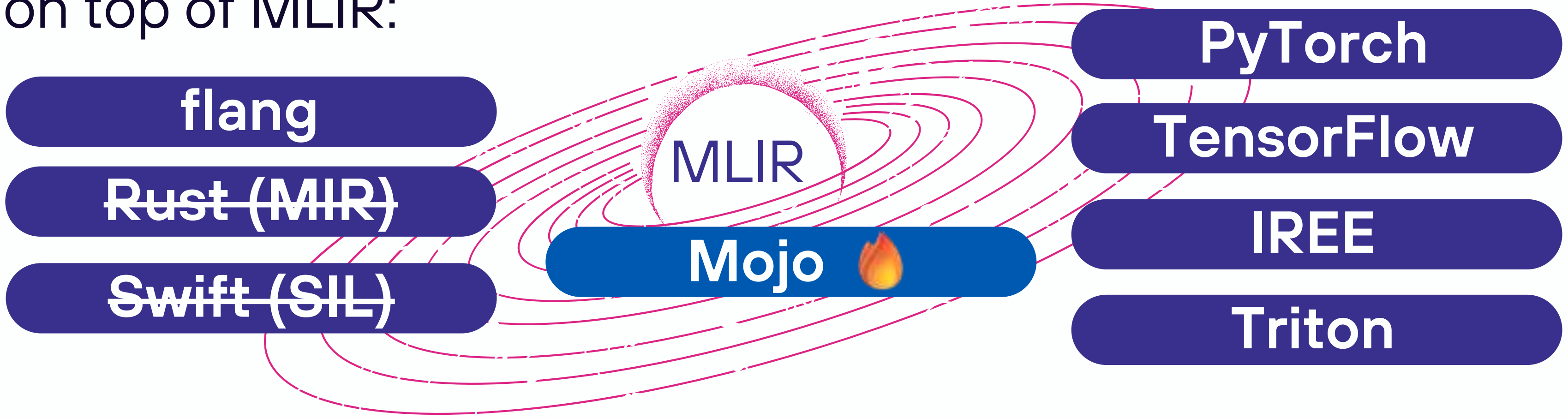
Backend

We will call JIT (Just-in-Time) engine from LLVM



(middle end) What is MLIR?

- Framework «Multi-Level IR» (2019, Chris Lattner & co)
  - High-level and low-level dialects
  - Different dialects in one IR (way to languages interop through IR? ~10+ years)
- LLVM project: llvm/llvm-project (folder mlir/)
- Built on top of MLIR:



Moving to MLIR currently:

clang IR (~30%) (my sharp estimation: it needs at least 2-3 years)

How such IR looks like? (1/3)

- IR is an SSA form (Static Single Assignment) in both MLIR and LLVM IR
  - each value has unique name/number, it is assigned only once
  - each value can be used many times

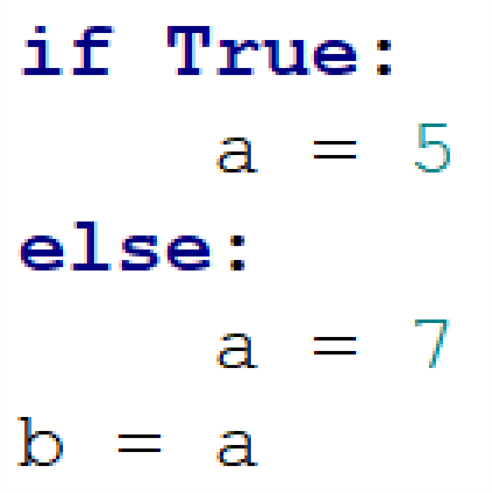
Any IR has three representations:

1. In-memory representation ~= C++ classes
2. Bytecode: serialized binary representation in a file (*.bc)
3. Textual IR (*.mlir): not just dump, it could be parsed into in-memory representation

- Functions can't contain in-place functions (like in Python)
- There is nothing similar to #include / import of other IR module (one IR -> one object file)

How such IR looks like? (2/3)

```
%result = dialect.operation @name (%operands) -> (result_type) { region }
```



```
if True:
    a = 5
else:
    a = 7
b = a
```

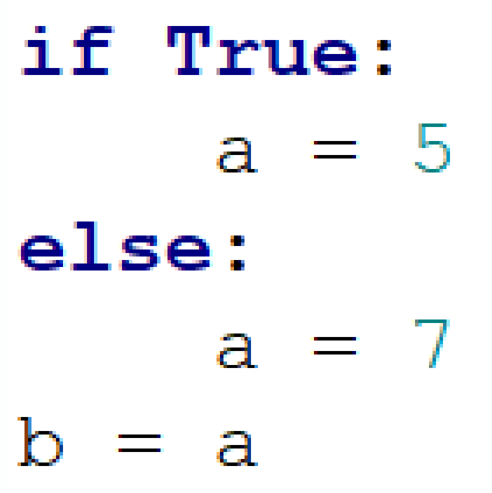
```
func.func @main() -> i64 {
    %0 = arith.constant 1 : i1
    %2 = scf.if %0 -> (i64) {
            %1 = arith.constant 5 : i64
            scf.yield %1 : i64
        } else {
            %1 = arith.constant 7 : i64
            scf.yield %1 : i64
        }
  func.return %2 : i64
}
```

How such IR looks like? (2/3)

```
%result = dialect.operation @name (%operands) -> (result_type) { region }
```

Some dialects:

- "scf" – structured control flow
- "func" – higher order functions
- "arith" – arithmetic operations



```
if True:
    a = 5
else:
    a = 7
b = a
```

```
func.func @main() -> i64 {
    %0 = arith.constant 1 : i1
    %2 = scf.if %0 -> (i64) {
            %1 = arith.constant 5 : i64
            scf.yield %1 : i64
        } else {
            %1 = arith.constant 7 : i64
            scf.yield %1 : i64
        }
  func.return %2 : i64
}
```

How such IR looks like? (3/3)

https://godbolt.org/z/j5WeEf8a4

```
llvm.func @main() -> i64 {
    %0 = llvm.mlir.constant(true) : i1
    llvm.cond_br %0, ^bb1, ^bb2
^bb1:
    %1 = llvm.mlir.constant(5 : i64) : i64
    llvm.br ^bb3(%1 : i64)
^bb2:
    %2 = llvm.mlir.constant(7 : i64) : i64
    llvm.br ^bb3(%2 : i64)
^bb3(%3: i64):
    llvm.br ^bb4
^bb4:
    llvm.return %3 : i64
}
```

```
func.func @main() -> i64 {
    %0 = arith.constant 1 : i1
    %2 = scf.if %0 -> (i64) {
            %1 = arith.constant 5 : i64
            scf.yield %1 : i64
        } else {
            %1 = arith.constant 7 : i64
            scf.yield %1 : i64
        }
  func.return %2 : i64
}
```

How different MLIR & LLVM IR?

MLIR диалект “llvm”

```
    %0 = llvm.mlir.constant(true) : i1
    llvm.cond_br %0, ^bb1, ^bb2
^bb1:
    %1 = llvm.mlir.constant(5 : i64) : i64
    llvm.br ^bb3(%1 : i64)
^bb2:
    %2 = llvm.mlir.constant(7 : i64) : i64
    llvm.br ^bb3(%2 : i64)
^bb3(%3: i64):
    llvm.br ^bb4
^bb4:
    llvm.return %3 : i64
```

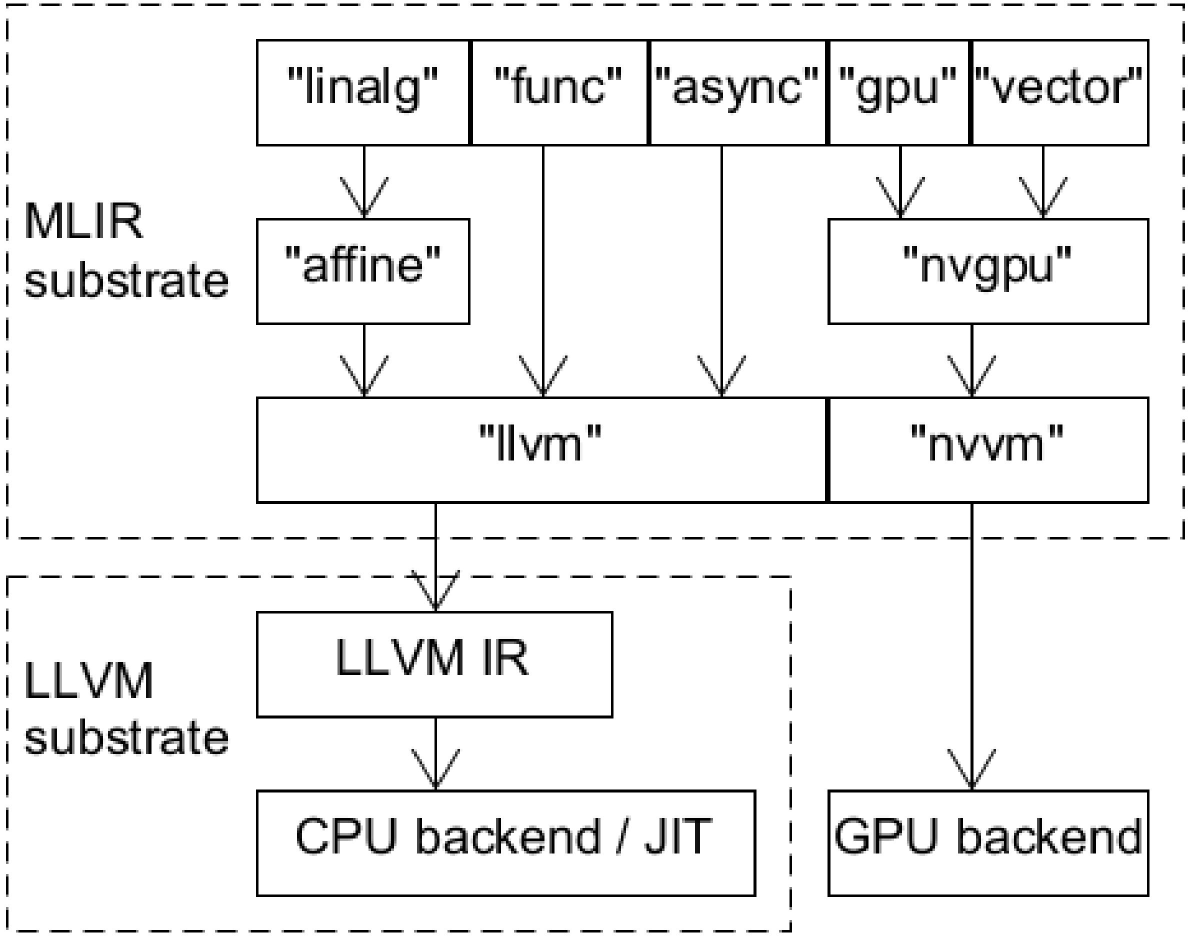
LLVM IR

```
    br i1 true, label %1, label %2
1:
    br label %3
2:
    br label %3
3:
    %4 = phi i64 [ 7, %2 ], [ 5, %1 ]
    br label %5
5:
    return %4 : i64
```

How to install MLIR?

- GitHub repository for this talk: vasily-v-ryabov/phdays24
  - Scripts, CMake files and source code
- On Ubuntu/Debian (WSL on Windows) there are apt packages
  - Already built LLVM and MLIR libraries and tools 17.x and 18.x
  - install.sh
- On Windows LLVM repo has to be cloned so that MLIR is built from sources
  - MLIR build takes up to 1 hour
  - Visual Studio 2022 must be installed (Community Edition is enough)
  - install.bat (run in VS2022 command prompt)

What happens inside MLIR?



- talk about substrates

1. **Generating high-level IR**
   (by visiting/walking Python AST)
2. **Lowering step by step**
   (Lowering/Conversion is a part of dialect)
3. **Translation to LLVM IR**
4. **Execution by LLVM's backend**

How will we generate IR?

“Toy” project: llvm/llvm-project: ./mlir/examples/toy/ (7 chapters)

- 1) frontend, **2) MLIR gen**, 3) dialect, 4) pass, 5) own lowering pass, **6) ->LLVM IR, 7) JIT**.

vasily-v-ryabov/phdays24 has 5 chapters:

1) The simplest IR generation	(22 lines in C++)
2) IR for variables assignments (with Python 3.9 frontend)	(+199 lines)
3) IR with conditions and type inference	(+59 lines)
4) Lowering and translation to LLVM IR (with options)	(+49 lines)
5) Using JIT engine	(+34 lines)
	Total: 363 lines

How will we generate IR?

vasily-v-ryabov/phdays24 has 5 chapters:

1) **The simplest IR generation**
2) IR for variables assignments (with Python 3.9 frontend)
3) IR with conditions and type inference
4) Lowering and translation to LLVM IR (with options)
5) Using JIT engine

Ya! The first code: main.cpp (1/3)

./phdays24/01_MLIR_gen/main.cpp

```
#include "mlir/IR/MLIRContext.h"
#include "mlir/IR/Verifier.h"

#include "mlir/Dialect/LLVMIR/LLVMDialect.h"

mlir::ModuleOp mlirGen(mlir::MLIRContext &context) {
  mlir::OpBuilder builder(&context);
  context.getOrLoadDialect<mlir::LLVM::LLVMDialect>();
  ...
  auto loc = builder.getUnknownLoc();

  auto module = mlir::ModuleOp::create(loc);
  builder.setInsertionPointToEnd(module.getBody());
  ...
```

Ya! The first code: main.cpp (2/3)

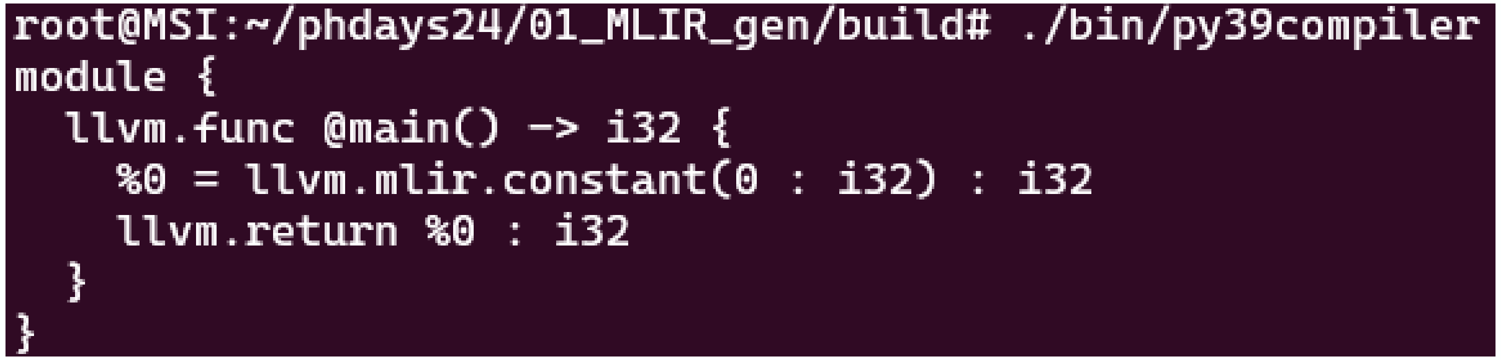
```
  ...
  // create function main()
  auto mainFuncType = mlir::LLVM::LLVMFunctionType::get(builder.getI32Type(), {});
  auto mainFunc = builder.create<mlir::LLVM::LLVMFuncOp>(loc, "main", mainFuncType);

  mlir::Block *entryBlock = mainFunc.addEntryBlock();
  builder.setInsertionPointToStart(entryBlock);

  // function body (return 0;)
  auto constOp = builder.create<mlir::LLVM::ConstantOp>(loc, builder.getI32Type(), 0);
  builder.create<mlir::LLVM::ReturnOp>(loc, constOp->getResult(0));
  return module;
}
```

Ya! The first code: main.cpp (3/3)

```
int main(int argc, char **argv) {
  mlir::MLIRContext context;
  mlir::OwningOpRef<mlir::ModuleOp> module = mlirGen(context);
  if (!module)
    return 1;
  if (mlir::failed(mlir::verify(*module))) {
    module->emitError("Module verification failed!");
    module->dump(); // dump incorrect IR anyway
    return 2;
  }
  module->dump(); // to stderr
  return 0;
} // 22 non-empty lines of code
```

Ya! The first code: main.cpp (3/3)

```cpp
int main(int argc, char **argv) {
  mlir::MLIRContext context;
  mlir::OwningOpRef<mlir::ModuleOp> module = mlirGen(context);
  if (!module)
    return 1;
  if (mlir::failed(mlir::verify(*module))) {
    module->emitError("Module verification failed!");
    module->dump(); // dump incorrect IR anyway
    return 2;
  }
  module->dump(); // to stderr
  return 0;
} // 22 non-empty lines of code
```



```
root@MSI:~/phdays24/01_MLIR_gen/build# ./bin/py39compiler
module {
  llvm.func @main() -> i32 {
    %0 = llvm.mlir.constant(0 : i32) : i32
    llvm.return %0 : i32
  }
}
```

How will we generate IR?

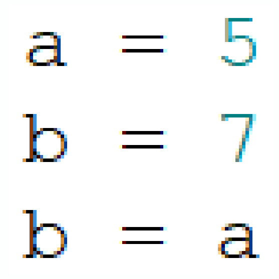
```
a = 5
b = 7
b = a
```

vasily-v-ryabov/phdays24 has 5 chapters:

1) The simplest IR generation
2) **IR for variables assignment** (with Python 3.9 frontend)
3) IR with conditions and type inference
4) Lowering and translation to LLVM IR (with options)
5) Using JIT engine

How to support variables? (1/4)

./phdays24/02_MLIR_gen_pyvars/include/py_ast.h



```
a = 5
b = 7
b = a
```

- class-wrapper for function RỳRắsșês AŞŢGsộņGîľê

```
class PyAST {
...
  bool parse_file(const char *name) { ... }
  mod_ty mod() { ... } // mod_ty is a pointer to C structure in Python.h
};
```

How to support variables? (2/4)

./phdays24/02_MLIR_gen_pyvars/include/MLIRGen.h

```
mlir::LogicalResult mlirGen(mod_ty pyModule) { ... }
mlir::LogicalResult mlirGen(asdl_seq *statements) { ... }
mlir::LogicalResult mlirGen(stmt_ty statement) {
  switch (statement->kind) {
    ...
    case Assign_kind: {
      auto valOrErr = mlirGen(/*expr_ty*/statement->v.Assign.value); // right side
      ... // left side
      expr_ty astTarget = (expr_ty)asdl_seq_GET(statement->v.Assign.targets, 0);
      ...
}
mlir::FailureOr<mlir::Value> mlirGen(expr_ty expr) { ... } // case Constant_kind:
```

How to support variables? (3/4)

```
a = 5
b = 7
b = a
```

- At the right side there is **name**, but where to get **mlir::Value** ?
- We need a symbol table! But it's not a simple std::hash_map...

```
#include "llvm/ADT/ScopedHashTable.h"
...
llvm::ScopedHashTable<llvm::StringRef, mlir::Value> symbolTable;
...
// at least one scope is required
llvm::ScopedHashTableScope<llvm::StringRef, mlir::Value> scope(symbolTable);
```

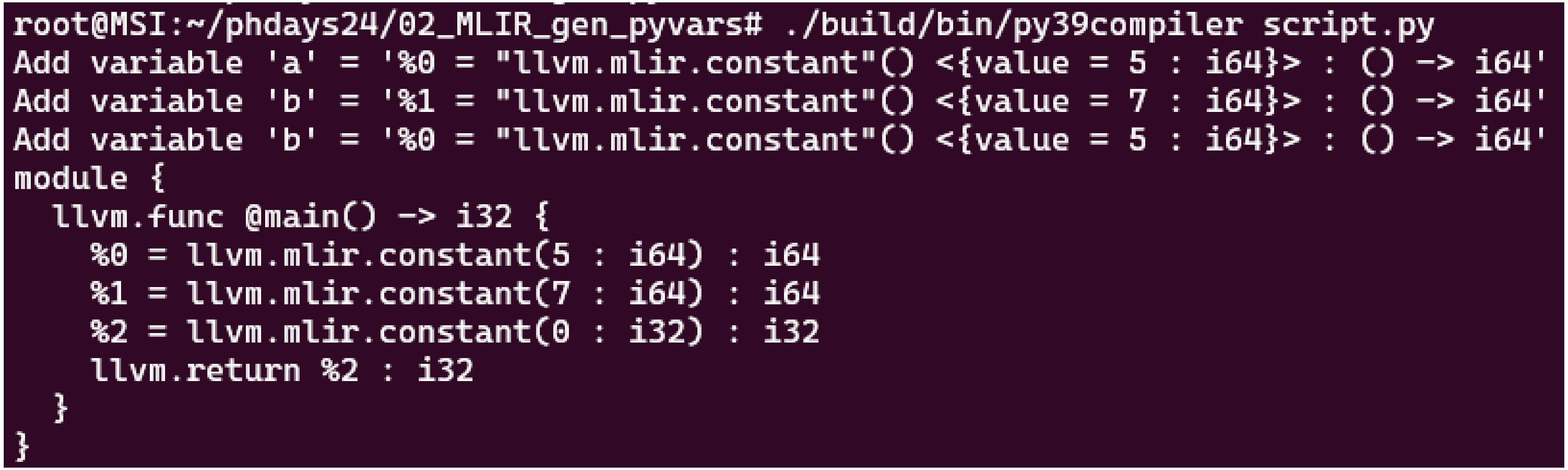
How to support variables? (4/4)

```
void defineVariable(llvm::StringRef name, mlir::Value value) {
  llvm::outs() << "Add variable '" << name << "' = '" << value << "'\n";
  llvm::MallocAllocator ma;
  symbolTable.insert(name.copy(ma), value);
}
mlir::FailureOr<mlir::Value> getVariable(mlir::Location loc, llvm::StringRef name) {
    auto value = symbolTable.lookup(name);
    if (value) // may be nullptr!
      return value;
    mlir::emitError(loc, "Variable '") << name << "' is not defined\n";
    return mlir::failure();
  }
```

- **MLIRGen.h**: ~200 lines of code, folder in the repo: **02_MLIR_gen_pyvars/**
- **py_ast.h**: ~50 lines of code

How to support variables? (4/4)

```
void defineVariable(llvm::StringRef name, mlir::Value value) {
  llvm::outs() << "Add variable '" << name << "' = '" << value << "'\n";
  llvm::MallocAllocator ma;
  symbolTable.insert(name.copy(ma), value);
}
```



```
root@MSI:~/phdays24/02_MLIR_gen_pyvars# ./build/bin/py39compiler script.py
Add variable 'a' = '%0 = "llvm.mlir.constant"() <{value = 5 : i64}> : () -> i64'
Add variable 'b' = '%1 = "llvm.mlir.constant"() <{value = 7 : i64}> : () -> i64'
Add variable 'b' = '%0 = "llvm.mlir.constant"() <{value = 5 : i64}> : () -> i64'
module {
  llvm.func @main() -> i32 {
    %0 = llvm.mlir.constant(5 : i64) : i64
    %1 = llvm.mlir.constant(7 : i64) : i64
    %2 = llvm.mlir.constant(0 : i32) : i32
    llvm.return %2 : i32
  }
}
```

What about globals?

1) Create LLVM::GlobalOp (“llvm.mlir.global”) – it is always a pointer
2) Get address by name: LLVM::AddressOfOp (“llvm.addressof”)
3) Load from pointer: LLVM::LoadOp (“llvm.load”)
4) Or store to pointer: LLVM::StoreOp (“llvm.store”)

- No scopes at all!
- “llvm.mlir.global” is similar to insert, but at runtime
- “llvm.addressof”+“llvm.load” are similar to lookup, but at runtime
- Functions are also globals
  - There is `mlir::SymbolTable globalTable(module);`
  - There is insert (not into IR) and lookup (useful to generate function body later)
  - There is module.lookupSymbol(name);
  - If “llvm.addressof” returned a function, it can be called by pointer (indirect call)

How will we generate IR?

```
if True:
    a = 5
else:
    a = 2
b = a
```

vasily-v-ryabov/phdays24 has 5 chapters:

1) The simplest IR generation
2) IR for variables assignment (with Python 3.9 frontend)
3) **IR with conditions and type inference**
4) Lowering and translation to LLVM IR (with options)
5) Using JIT engine

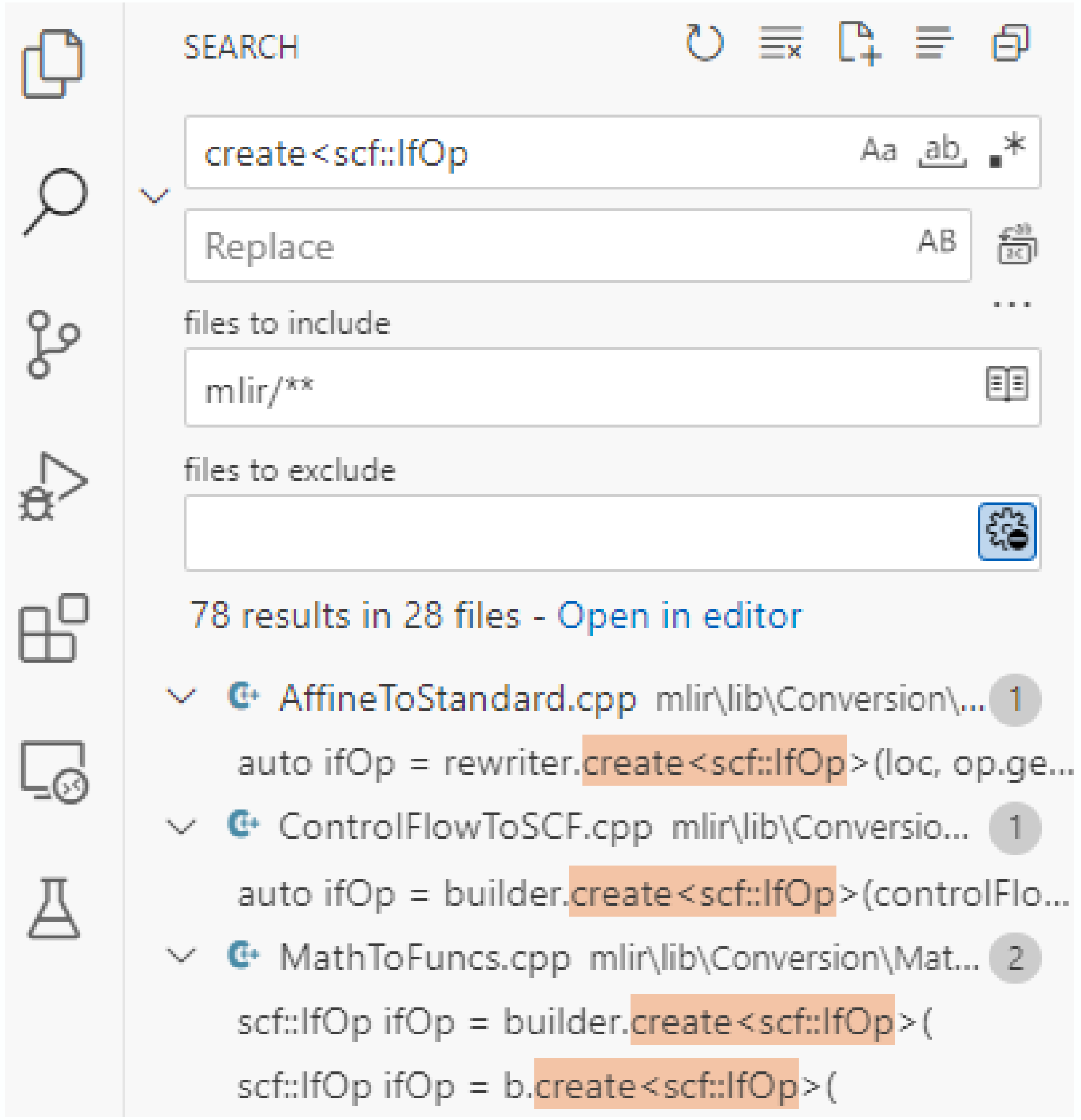
How will we add conditions? (1/5)

```
#include "mlir/Dialect/SCF/IR/SCF.h"
#include "mlir/Dialect/ControlFlow/IR/ControlFlow.h"
...
context.getOrLoadDialect<mlir::scf::SCFDialect>();
context.getOrLoadDialect<mlir::cf::ControlFlowDialect>();
...

auto ifOp = builder.create<mlir::scf::IfOp>(loc, ...); // parameters?
```

```
if True:
    a = 5
else:
    a = 2
b = a
```

How to find builders

- No builders' info in the dialect docs!
- We have to find them in the MLIR sources


SEARCH

create<scf::IfOp

Replace

files to include

mlir/**

files to exclude

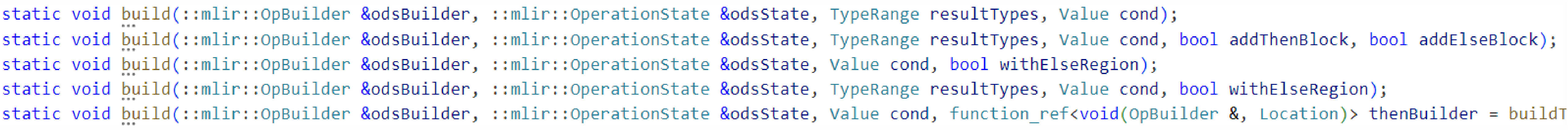
78 results in 28 files - Open in editor

```
AffineToStandard.cpp  mlir\lib\Conversion\...  1
    auto ifOp = rewriter.create<scf::IfOp>(loc, op.ge...
ControlFlowToSCF.cpp  mlir\lib\Conversio...  1
    auto ifOp = builder.create<scf::IfOp>(controlFlo...
MathToFuncs.cpp  mlir\lib\Conversion\Mat...  2
    scf::IfOp ifOp = builder.create<scf::IfOp>(
    scf::IfOp ifOp = b.create<scf::IfOp>(
```

How to find builders

- No builders' info in the dialect docs!
- We have to find them in the MLIR sources

- If we go into class scf::IfOp, ...

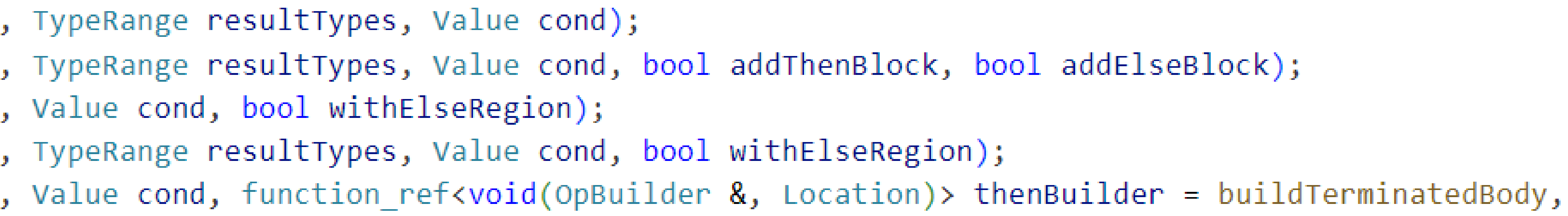


```
static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, TypeRange resultTypes, Value cond);
static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, TypeRange resultTypes, Value cond, bool addThenBlock, bool addElseBlock);
static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, Value cond, bool withElseRegion);
static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, TypeRange resultTypes, Value cond, bool withElseRegion);
static void build(::mlir::OpBuilder &odsBuilder, ::mlir::OperationState &odsState, Value cond, function_ref<void(OpBuilder &, Location)> thenBuilder = buildT
```

How to find builders

- No builders' info in the dialect docs!
- We have to find them in the MLIR sources

- If we go into class scf::IfOp, ...



```
, TypeRange resultTypes, Value cond);
, TypeRange resultTypes, Value cond, bool addThenBlock, bool addElseBlock);
, Value cond, bool withElseRegion);
, TypeRange resultTypes, Value cond, bool withElseRegion);
, Value cond, function_ref<void(OpBuilder &, Location)> thenBuilder = buildTerminatedBody,
```

How will we add conditions? (2/5)

```
auto valueOrError = mlirGen(statement->v.If.test);
if (mlir::failed(valueOrError))
  return mlir::failure();
auto condition = valueOrError.value(); // TODO: check type is i1 (integer of 1 bit)
auto ifOp = builder.create<mlir::scf::IfOp>( // it calls inferReturnTypes(...)
    loc, condition,
    /*thenBuilder=*/[&](mlir::OpBuilder &b, mlir::Location loc) {

      b.create<mlir::scf::YieldOp>(loc /*returned values*/);
    },
    /*elseBuilder=*/[&](mlir::OpBuilder& b, mlir::Location loc) {

      b.create<mlir::scf::YieldOp>(loc /*returned values*/);
    }
);
```

How will we add conditions? (2/5)

```
auto valueOrError = mlirGen(statement->v.If.test);
if (mlir::failed(valueOrError))
  return mlir::failure();
auto condition = valueOrError.value(); // TODO: check type is i1 (integer of 1 bit)
auto ifOp = builder.create<mlir::scf::IfOp>( // it calls inferReturnTypes(...)
    loc, condition,
    /*thenBuilder=*/[&](mlir::OpBuilder &b, mlir::Location loc) {
      mlirGen(statement->v.If.body); // Assign_kind AST-nodes can be inside
      b.create<mlir::scf::YieldOp>(loc /*returned values*/);
    },
    /*elseBuilder=*/[&](mlir::OpBuilder& b, mlir::Location loc) {
      mlirGen(statement->v.If.orelse); // Assign_kind AST-nodes can be inside
      b.create<mlir::scf::YieldOp>(loc /*returned values*/);
    }
);
```

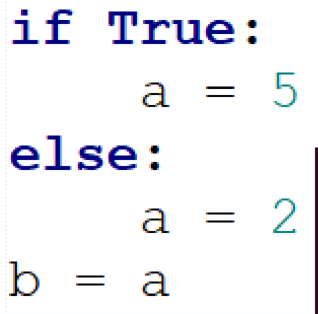
How will we add conditions? (3/5)

```
std::stack<std::set<llvm::StringRef>> ifElseVariables;
llvm::MallocAllocator ma; // for copying llvm::StringRef’s
...
void defineVariable(llvm::StringRef name, mlir::Value value) {
  symbolTable.insert(name.copy(ma), value);
  if (!ifElseVariables.empty())
    ifElseVariables.top().insert(name.copy(ma));
}
```

How will we add conditions? (4/5)

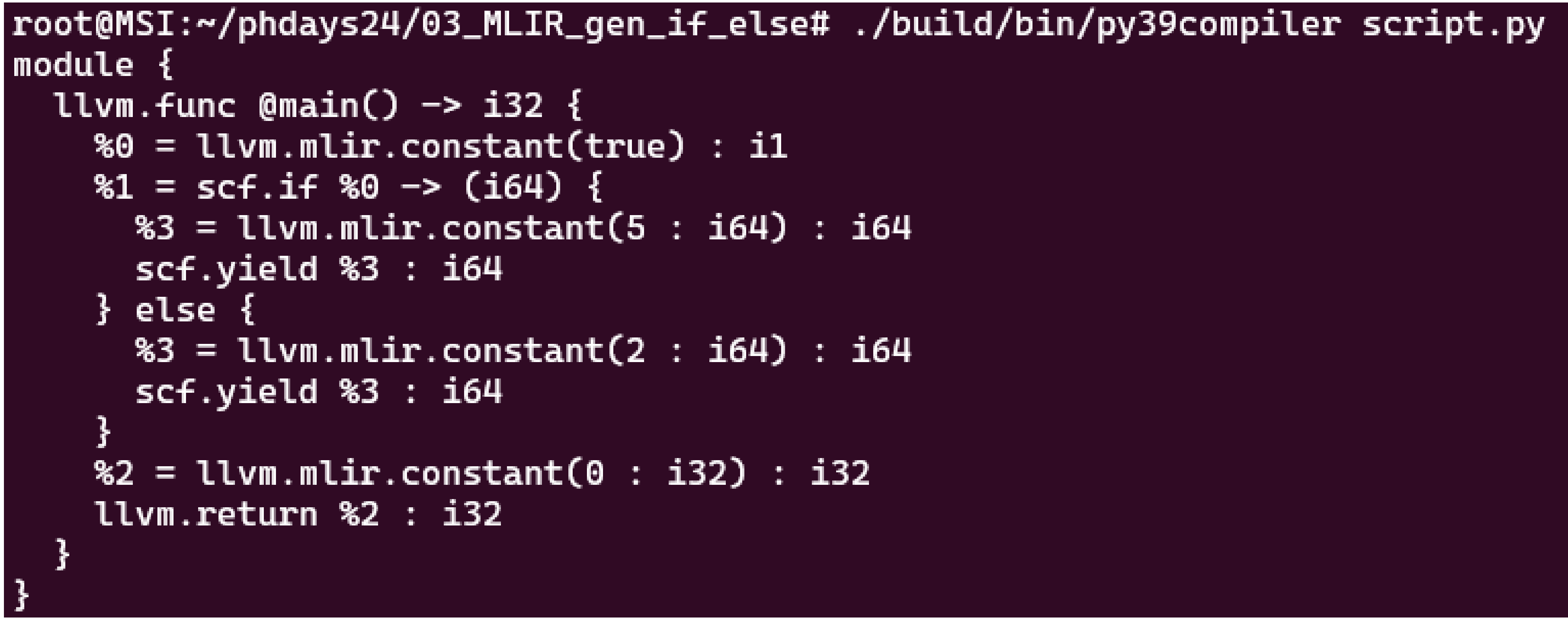
```
/*thenBuilder=*/
[&](mlir::OpBuilder &b, mlir::Location loc) {
  ifElseVariables.push(std::set<llvm::StringRef>());
  result = mlirGen(statement->v.If.body);
  auto varsSet = ifElseVariables.top();
  llvm::SmallVector<mlir::Value> returnValues;
  for (auto it = varsSet.begin(); it != varsSet.end(); it++) {
    auto value = symbolTable.lookup(*it);
    if (value)
      returnValues.push_back(value);
  }
  ifElseVariables.pop();
  b.create<mlir::scf::YieldOp>(loc, returnValues);
},
```

How will we add conditions? (5/5)



```python
if True:
    a = 5
else:
    a = 2
b = a
```

(slide 7 –> slide 35) == 🔥



```
root@MSI:~/phdays24/03_MLIR_gen_if_else# ./build/bin/py39compiler script.py
module {
  llvm.func @main() -> i32 {
    %0 = llvm.mlir.constant(true) : i1
    %1 = scf.if %0 -> (i64) {
      %3 = llvm.mlir.constant(5 : i64) : i64
      scf.yield %3 : i64
    } else {
      %3 = llvm.mlir.constant(2 : i64) : i64
      scf.yield %3 : i64
    }
    %2 = llvm.mlir.constant(0 : i32) : i32
    llvm.return %2 : i32
  }
}
```

How will we generate IR?

vasily-v-ryabov/phdays24 has 5 chapters:

1) The simplest IR generation
2) IR for variables assignment (with Python 3.9 frontend)
3) IR with conditions and type inference
4) **Lowering and translation to LLVM IR** (with options)
5) Using JIT engine

How to run lowering

1. "scf" – structured control flow
2. "cf" – control flow
3. "llvm"

- mlir-opt-18 --help | grep to-llvm
- mlir-opt-18 --convert-scf-to-cf input.mlir
- mlir-opt-18 --convert-cf-to-llvm input.mlir

How to run lowering

1. “scf” – structured control flow
2. “cf” – control flow
3. “llvm”
4. LLVM IR

- mlir-opt-18 --help | grep to-llvm
- mlir-opt-18 --convert-scf-to-cf input.mlir
- mlir-opt-18 --convert-cf-to-llvm input.mlir
- mlir-translate-18 --mlir-to-llvmir input.mlir

How to run lowering

1. “scf” – structured control flow
2. “cf” – control flow
3. “llvm”
4. LLVM IR
5. Execute on JIT engine

- `mlir-opt-18 --help | grep to-llvm`
- `mlir-opt-18 --convert-scf-to-cf input.mlir`
- `mlir-opt-18 --convert-cf-to-llvm input.mlir`
- `mlir-translate-18 --mlir-to-llvmir input.mlir`
- `lli output.ll` or `mlir-cpu-runner input.mlir`

How to run lowering

1. “scf” – structured control flow	• mlir-opt-18 --help | grep to-llvm
2. “cf” – control flow	• mlir-opt-18 --convert-scf-to-cf input.mlir
3. “llvm”	• mlir-opt-18 --convert-cf-to-llvm input.mlir
4. LLVM IR	• mlir-translate-18 --mlir-to-llvmir input.mlir
5. Execute on JIT engine	• lli output.ll or mlir-cpu-runner input.mlir
(what if it crashed? w/ huge IR!)	• mlir-opt-18 --mlir-pass-pipeline-crash-reproducer=<output_filepath.mlir>

How to run lowering

1. "scf" – structured control flow
2. "cf" – control flow
3. "llvm"
4. LLVM IR
5. Execute on JIT engine

- mlir-opt-18 --help | grep to-llvm
- mlir-opt-18 --convert-scf-to-cf input.mlir
- mlir-opt-18 --convert-cf-to-llvm input.mlir
- mlir-translate-18 --mlir-to-llvmir input.mlir
- lli output.ll or mlir-cpu-runner input.mlir

(what if it crashed? w/ huge IR!)

- mlir-opt-18 --mlir-pass-pipeline-crash-reproducer=<output_filepath.mlir>

(how to get back to MLIR?)

- 4. LLVM IR → 3. MLIR "llvm"

- clang -S -emit-llvm foo.c >foo.ll
- mlir-translate-18 --import-llvm foo.ll >foo.mlir

How to write lowering

```
#include "mlir/Conversion/SCFToControlFlow/SCFToControlFlow.h"
#include "mlir/Conversion/ControlFlowToLLVM/ControlFlowToLLVM.h“
#include "mlir/Pass/Pass.h"
#include "mlir/Pass/PassManager.h"
...
int main(int argc, char **argv) {
  ...
  mlir::PassManager passes(&context);
  passes.addPass(mlir::createConvertSCFToCFPass());
  passes.addPass(mlir::createConvertControlFlowToLLVMPass());
  if (mlir::failed(passes.run(module.get())))
    return 5;
}
```

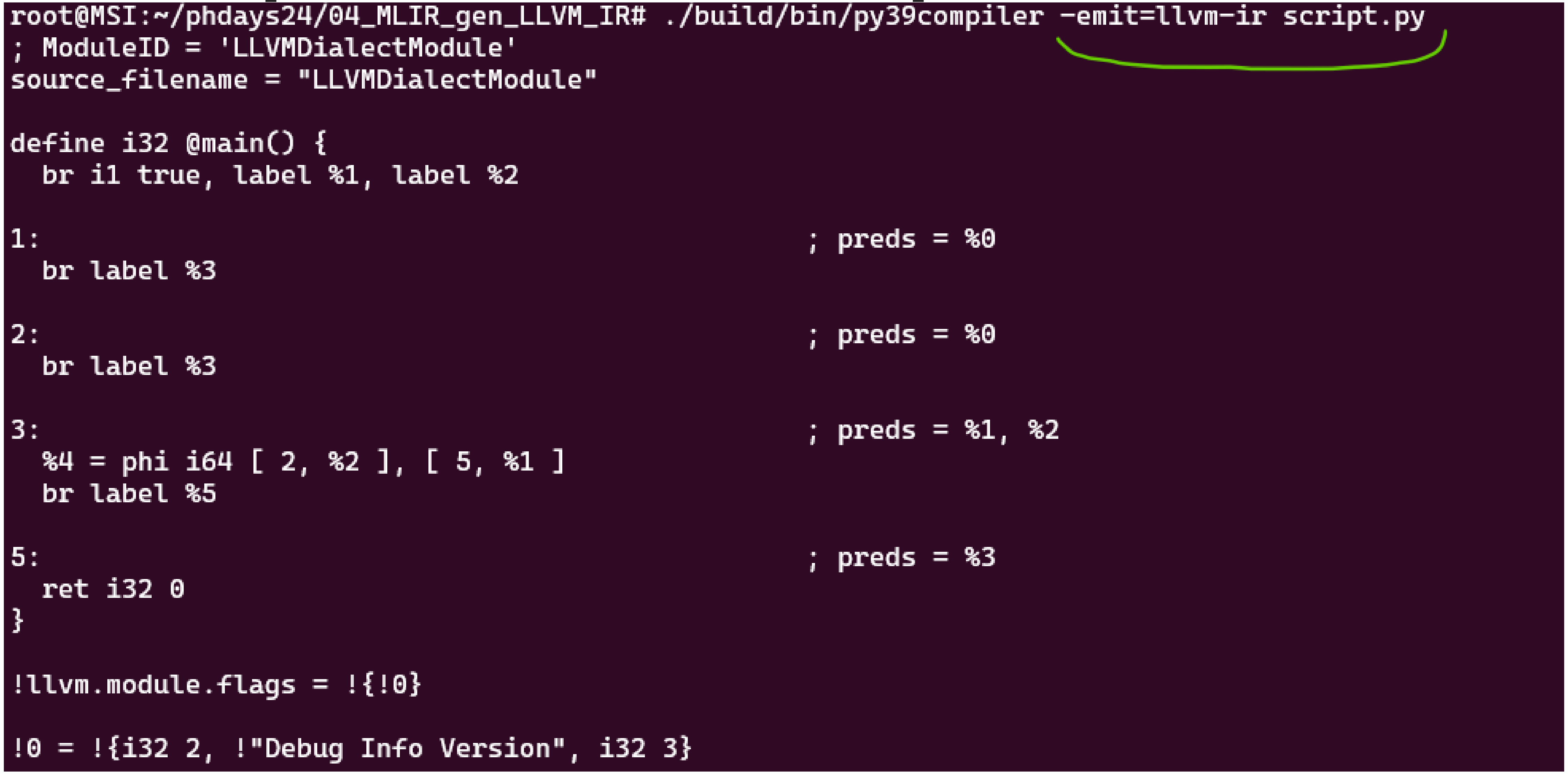
How to translate to LLVM IR

```
#include "mlir/Target/LLVMIR/Dialect/Builtin/BuiltinToLLVMIRTranslation.h"
#include "mlir/Target/LLVMIR/Dialect/LLVMIR/LLVMToLLVMIRTranslation.h"
#include "mlir/Target/LLVMIR/Export.h"
...
int main(int argc, char **argv) {
  ...
  mlir::registerBuiltinDialectTranslation(*module->getContext());
  mlir::registerLLVMDialectTranslation(*module->getContext());

  llvm::LLVMContext llvmContext;
  auto llvmModule = mlir::translateModuleToLLVMIR(*module, llvmContext);
  llvm::errs() << *llvmModule << "\n"; // dump LLVM IR
}
```

How to parse CmdLine options

```
#include "llvm/Support/CommandLine.h"
...
enum Action { DumpMLIR, DumpLLVM, DumpLLVMIR };
namespace cl = llvm::cl;
static cl::opt<enum Action> emitAction("emit", cl::desc("Select the output"),
    cl::values(clEnumValN(DumpMLIR, "mlir", "output the MLIR dump")),
    cl::values(clEnumValN(DumpLLVM, "llvm", "dump the MLIR \"llvm\" dialect")),
    cl::values(clEnumValN(DumpLLVMIR, "llvm-ir", "output the LLVM IR dump")));

static cl::opt<std::string> srcPy(cl::Positional, cl::desc("<input .py file>"),
                                  cl::init("-"), cl::value_desc("filename"));
int main(int argc, char **argv) {
  cl::ParseCommandLineOptions(argc, argv, "Python 3.9 demo compiler\n");
...
```

How to parse CmdLine options



```
root@MSI:~/phdays24/04_MLIR_gen_LLVM_IR# ./build/bin/py39compiler -emit=llvm-ir script.py
; ModuleID = 'LLVMDialectModule'
source_filename = "LLVMDialectModule"

define i32 @main() {
  br i1 true, label %1, label %2

1:                                                ; preds = %0
  br label %3

2:                                                ; preds = %0
  br label %3

3:                                                ; preds = %1, %2
  %4 = phi i64 [ 2, %2 ], [ 5, %1 ]
  br label %5

5:                                                ; preds = %3
  ret i32 0
}

!llvm.module.flags = !{!0}

!0 = !{i32 2, !"Debug Info Version", i32 3}
```

How will we generate IR?

vasily-v-ryabov/phdays24 has 5 chapters:

1) The simplest IR generation
2) IR for variables assignment (with Python 3.9 frontend)
3) IR with conditions and type inference
4) Lowering and translation to LLVM IR (with options)
5) **Using JIT engine**

How to integrate JIT engine

```
#include "mlir/ExecutionEngine/ExecutionEngine.h"
#include "mlir/ExecutionEngine/OptUtils.h"
#include "llvm/ExecutionEngine/Orc/JITTargetMachineBuilder.h"
#include "llvm/Support/TargetSelect.h"
```

```
llvm::InitializeNativeTarget();
llvm::InitializeNativeTargetAsmPrinter();
mlir::ExecutionEngineOptions engineOptions;
auto maybeEngine = mlir::ExecutionEngine::create(*module, engineOptions);
auto &engine = maybeEngine.get();
...
llvm::SmallVector<void *> argsAndReturn; // int32_t main() without argc, argv[]
int32_t exitCode; argsAndReturn.push_back(&exitCode); // address of return value
auto invocationResult = engine->invokePacked("main", argsAndReturn);
```

What else has the JIT engine?

- Hot code detection and its re-optimization
  - example: -O0 is the default level, -O2 is only for hot code
- JIT callback: runtime can call a compiler back (runtime -> compiler -> runtime)
  - example is somewhere in ./llvm/unittests/ExecutionEngine/Orc/
- Support of the debugger for your language (ORCDebugging library)
- Materialization layers (used by Mojo 🔥 a lot)
- https://www.llvm.org/docs/ORCv2.html

What is absent in “scf” dialect?

- “scf.break”
- “scf.continue”

- early return
  - “func.return”: HasParent<FuncOp> (terminator only for “func.func”)
  - “llvm.return”: it will be lowered as is, but we need “llvm.br” to a final block

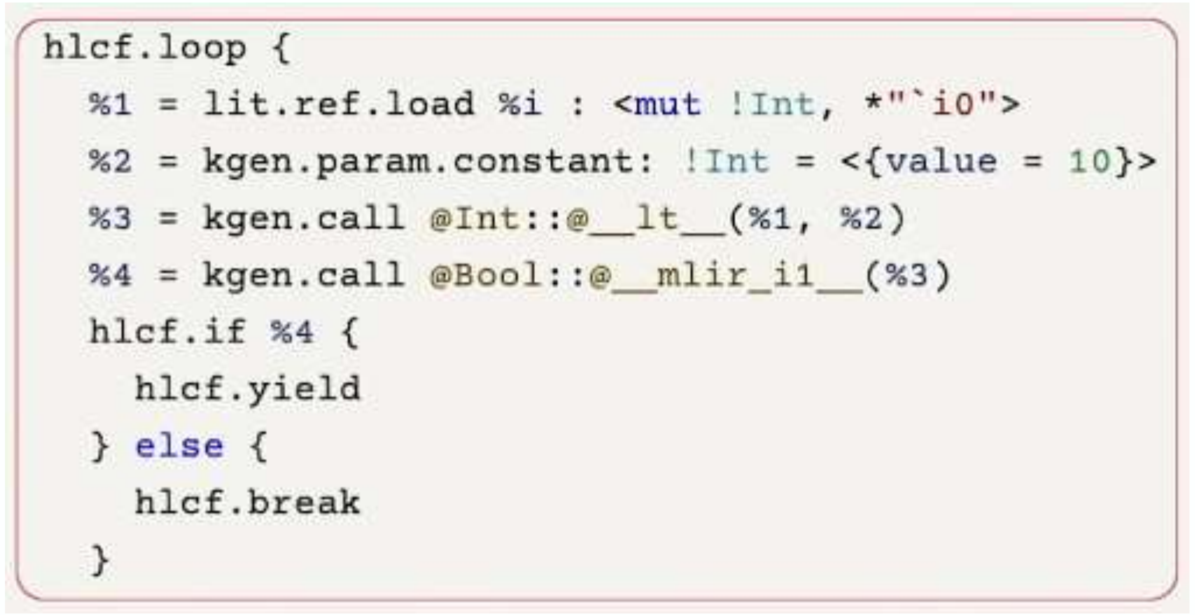
- But we can:
  - use dialect “cf”: “cf.br” и “cf.cond_br” (or “llvm.br” and “llvm.cond_br”)

What's up with Mojo 🔥 ?

Mojo 🔥 has dialect "hlcf": [(YouTube) 2023 LLVM Dev mtg Mojo (time=17:00)]()

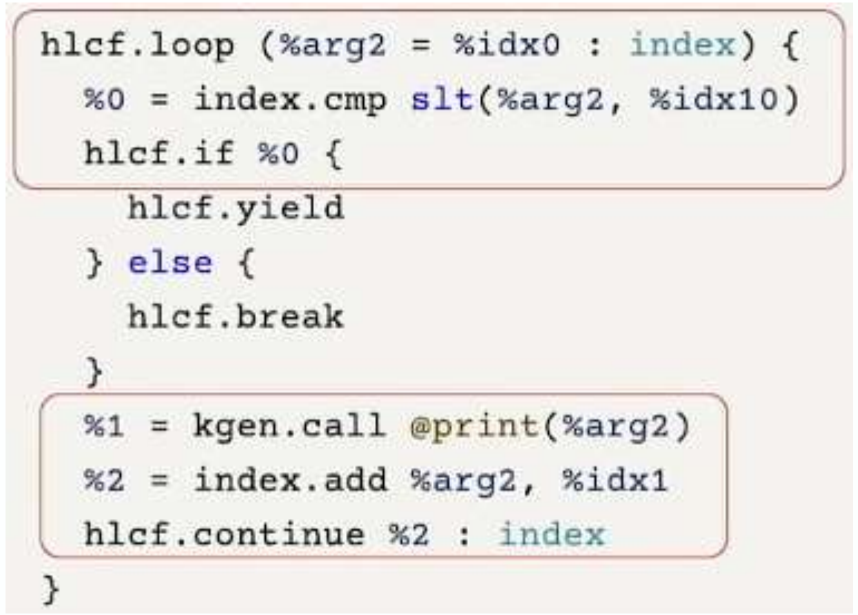
(High-Level Control Flow)

- "hlcf.break"



```
hlcf.loop {
  %1 = lit.ref.load %i : <mut !Int, *"`i0">
  %2 = kgen.param.constant: !Int = <{value = 10}>
  %3 = kgen.call @Int::@__lt__(%1, %2)
  %4 = kgen.call @Bool::@__mlir_i1__(%3)
  hlcf.if %4 {
    hlcf.yield
  } else {
    hlcf.break
  }
```

"hlcf.continue"



```
hlcf.loop (%arg2 = %idx0 : index) {
  %0 = index.cmp slt(%arg2, %idx10)
  hlcf.if %0 {
    hlcf.yield
  } else {
    hlcf.break
  }
  %1 = kgen.call @print(%arg2)
  %2 = index.add %arg2, %idx1
  hlcf.continue %2 : index
}
```

- early return: it is supported, but without many details

What's up with clangIR?

clang/include/clang/CIR/Dialect/IR/CIROps.td#L797

- "cir.break"
- "cir.continue"

- "cir.return" (early return): clang/include/clang/CIR/Dialect/IR/CIROps.td#L540
  - ParentOneOf<["FuncOp", "ScopeOp", "IfOp", "SwitchOp", "DoWhileOp", "WhileOp", "ForOp"]

What is absent in “func” dialect?

- “func.invoke”
  - it is a function call, which may raise an exception
  - there is “llvm.invoke” (in theory the “llvm” dialect should be enough for exception handling, but a lot of code is required)
- But “func.func” can return multiple values (“llvm.func” can return only one)
  - the workaround is to return “llvm.struct” type (example: “llvm.struct “A”<i64,i64,f64>”)
- If we look into clangIR operations (CIROps.td):
  - “cir.try_call” (cir::TryCallOp), “cir.try” (cir::TryOp)
  - “cir.catch” (cir::CatchOp), “cir.catch_param” (cir::CatchParamOp)
  - “cir.alloc_exception” (cir::AllocExceptionOp), “cir.throw” (cir::ThrowOp)
  - “cir.stack_save” (cir::StackSaveOp), “cir.stack_restore” (cir::StackRestoreOp)
  - (CIRTypes.td) see type “cir.eh.info”
- Class support is required for exceptions (dialect “cir” supports classes)

Which dialects are used by Mojo 🔥 ?

- Upstream dialects “llvm” и “index”

- “hlcf” is High level control flow
- “mosh” is “Mojo 🔥 Shape” dialect (shape is a size of vector, matrix or tensor)
- “kgen” is a meta dialect, it has **ElaborationPass** which do the most of job!
  - for functions with [type params] like this: var a = func**[t1, t2]**(arg1, arg2)
  - for structures with [type params] similar to C++ templates
  - for compile time introspection

What is unique in Mojo 🔥 ?

- **AST** (Abstract Syntax Tree) **is not built** at all!
  - Source code parser immediately generates an MLIR IR
  - MLIR has constant folding hooks by design (typical AST level optimization)
  - Interview with Chris Lattner (May 2024)

- Standard module TargetInfo (low level) is used for loop unrolling at high level

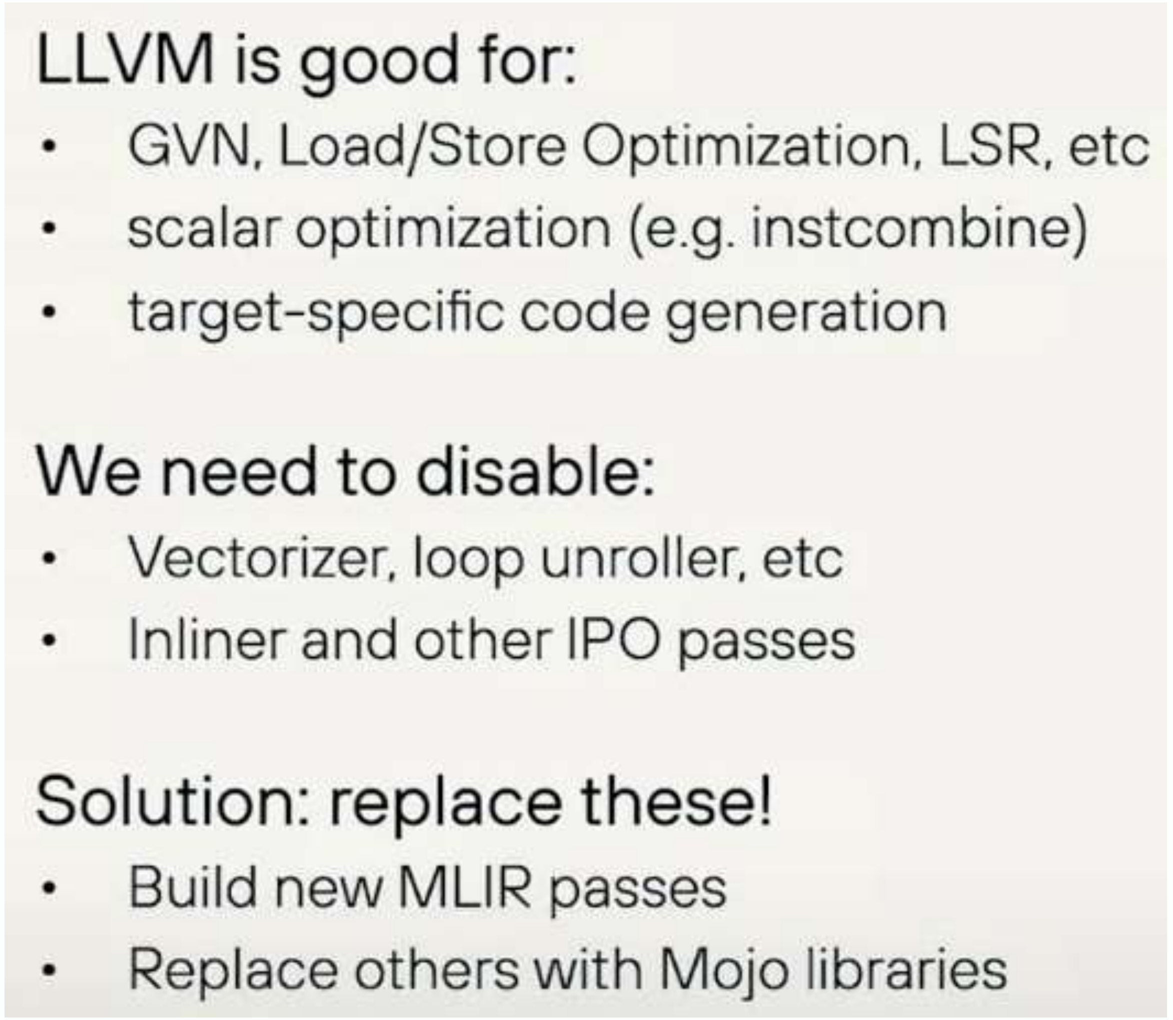
- JIT engine ORC (from LLVM) is used **at all stages** for adaptive compilation
  - both just-in-time (REPL), **and ahead-of-time**!
  - materialization layers are created

What Mojo 🔥 took from LLVM?

[(c) Talk about Mojo in 2023 (at 27:15)]

80th level of parallelism (at 27:40):

- One LLVMContext per function



LLVM is good for:

- GVN, Load/Store Optimization, LSR, etc
- scalar optimization (e.g. instcombine)
- target-specific code generation

We need to disable:

- Vectorizer, loop unroller, etc
- Inliner and other IPO passes

Solution: replace these!

- Build new MLIR passes
- Replace others with Mojo libraries

Why not learn Mojo 🔥 yet?

The language is changing regularly:

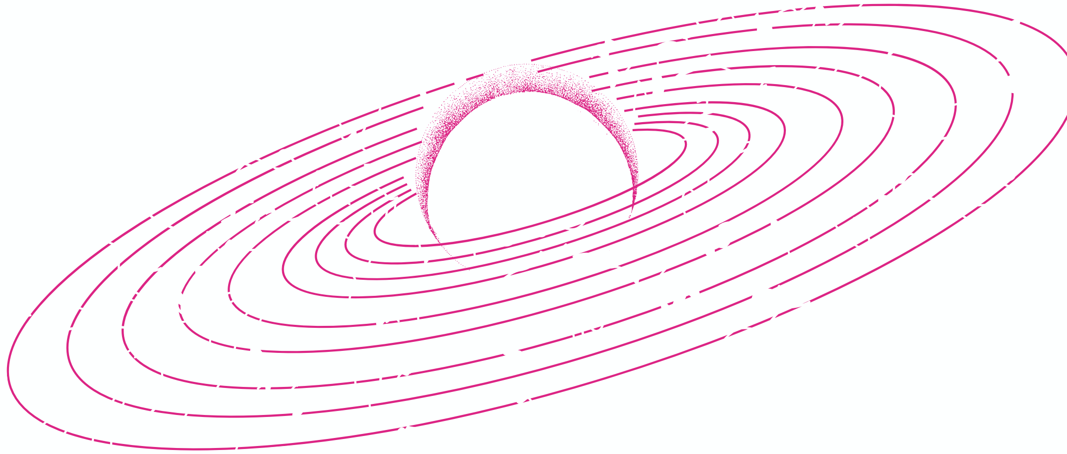
- let variables were removed recently
- dynamic types at runtime are on re-design

The MLIR dialects are not open:

- “kgen”, “hlcf”, … (not mature?)

Priorities (in my opinion at the moment):

- P1: AI engine
- P2: MLIR frontend (the language for compiler engineers)
- P3: Python superset (superset++, i.e. much wider than Python)



Conclusion: learn MLIR and JIT engine ORC => fast learn Mojo 🔥

What else to watch?

🡐 playlist

In Russian:

- Constantine Vladimirov's lecture about LLVM IR (2019) – GEP, load, alloca

In English:

- Extending Dominance to MLIR Regions (2023)
- MLIR Dialect Design (EuroLLVM 2023) – dialects classification
- What's new in MLIR? (2023 vs 2019)
- JITLink: Native Windows JITing in LLVM & ORCv2 - LLVM JIT APIs Deep Dive

- www.youtube.com/@LLVMPROJ – LLVM Dev meetings & EuroLLVM

@vasily_v_ryabov

vasily.v.ryabov@gmail.com

github.com/vasily-v-ryabov

@vasily_v_ryabov

phd 2 Positive Hack Days Fest

от ■ positive technologies

Thank you!

OK build MLIR (backup slide)

```
git clone --depth 1 -b llvmorg-18.1.5 https://github.com/llvm/llvm-project.git

cd ./llvm-project/ && mkdir build && cd build/

pip install -U cmake ninja

cmake -G Ninja ../llvm/ -DLLVM_ENABLE_PROJECTS="mlir" -DLLVM_ENABLE_ASSERTIONS=ON -DCMAKE_BUILD_TYPE=Debug -DLLVM_TARGETS_TO_BUILD="Native"

cmake --build . --target check-mlir  # (take cap of tea, wait for 1h)

# done!


# on Windows (before building!) need to run regedit, go to path:

# HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem\

# set LongPathsEnabled = 1 and reboot the OS!

# Instead -G Ninja we use something like -G "Visual Studio 17 2022"
```

What is the hardest to me?

Middle end

Type inference – undecidable for many type systems
(in dataset ManyType4Py only 60-70% types are inferred)

Dialect design is an art (and, of course, TableGen and C++)