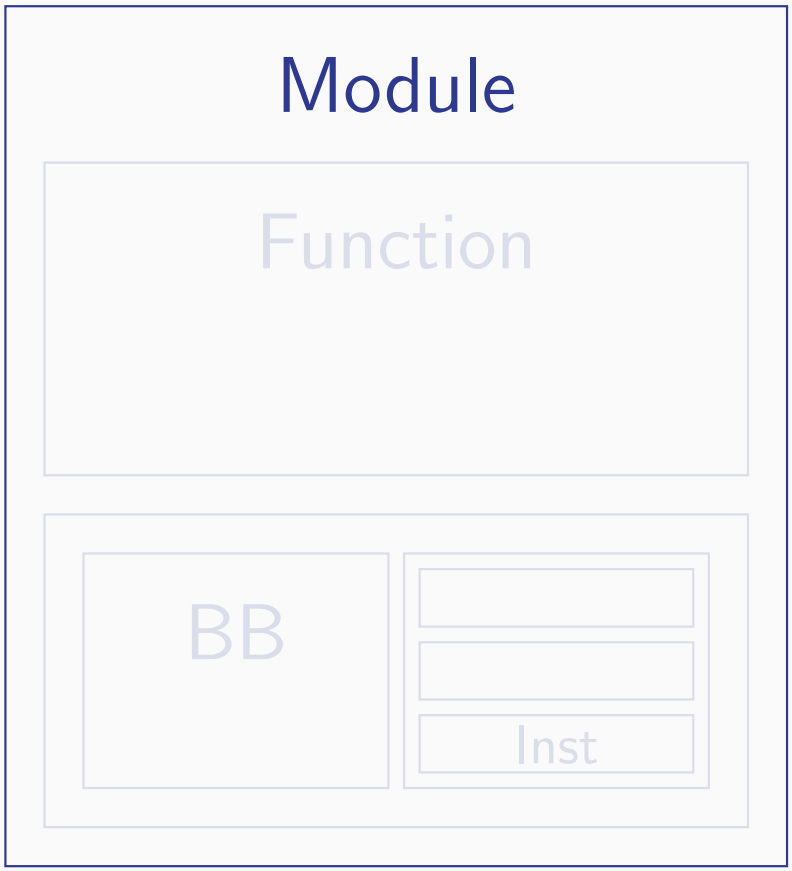
Building, Testing and Debugging a Simple out-of-tree LLVM Pass

October 29, 2015, LLVM Developers' Meeting

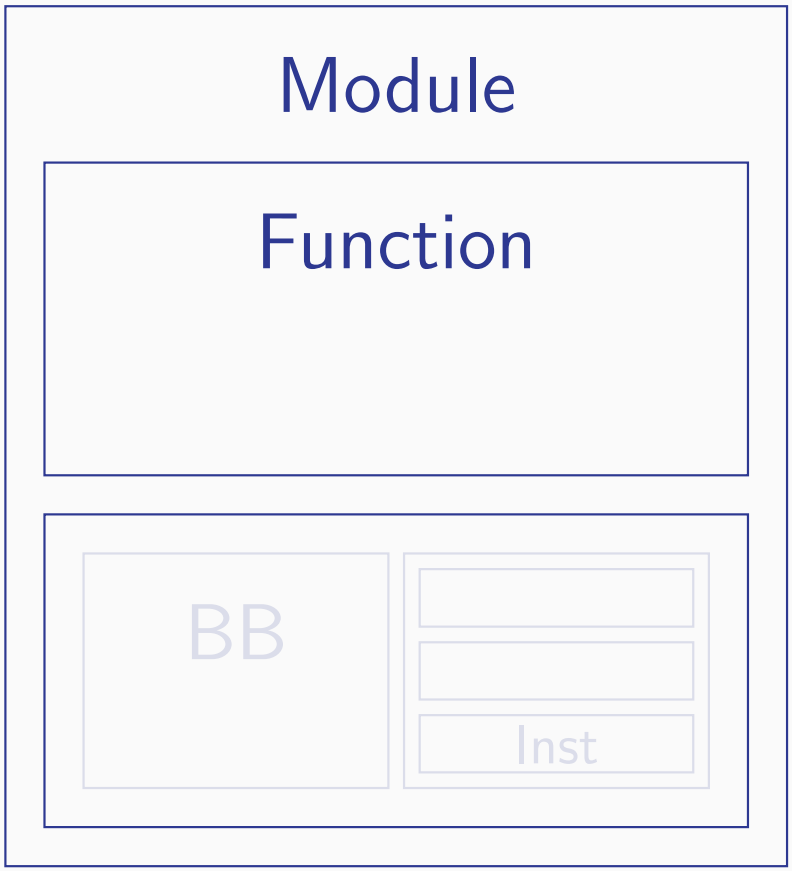
LLVM 3.7 — Resources

https://github.com/quarkslab/
llvm-dev-meeting-tutorial-2015

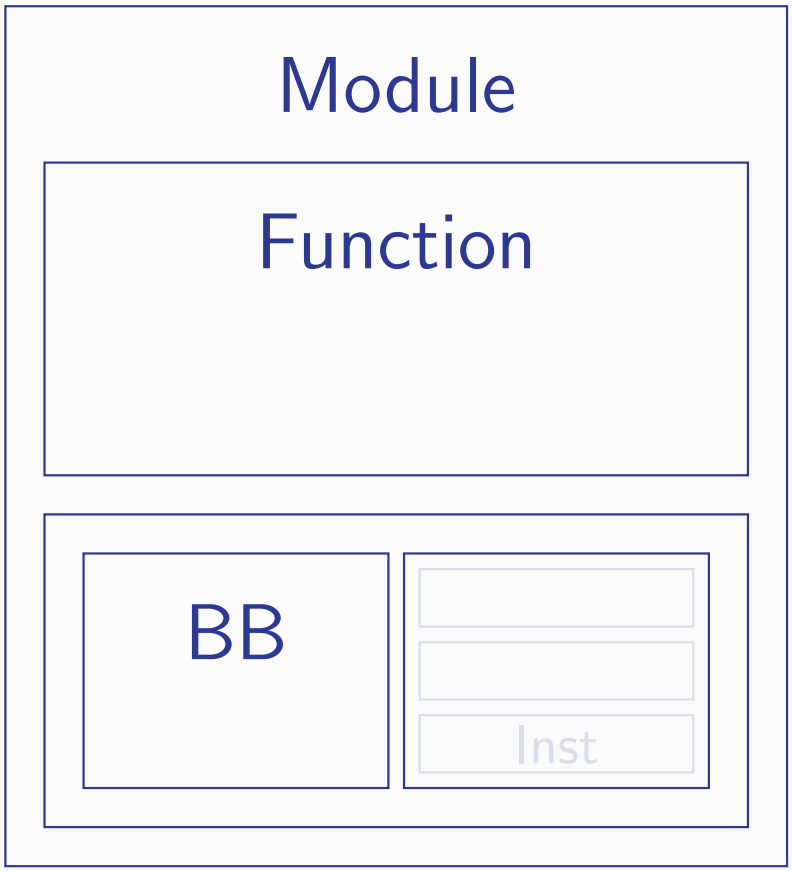
Instruction Booklet



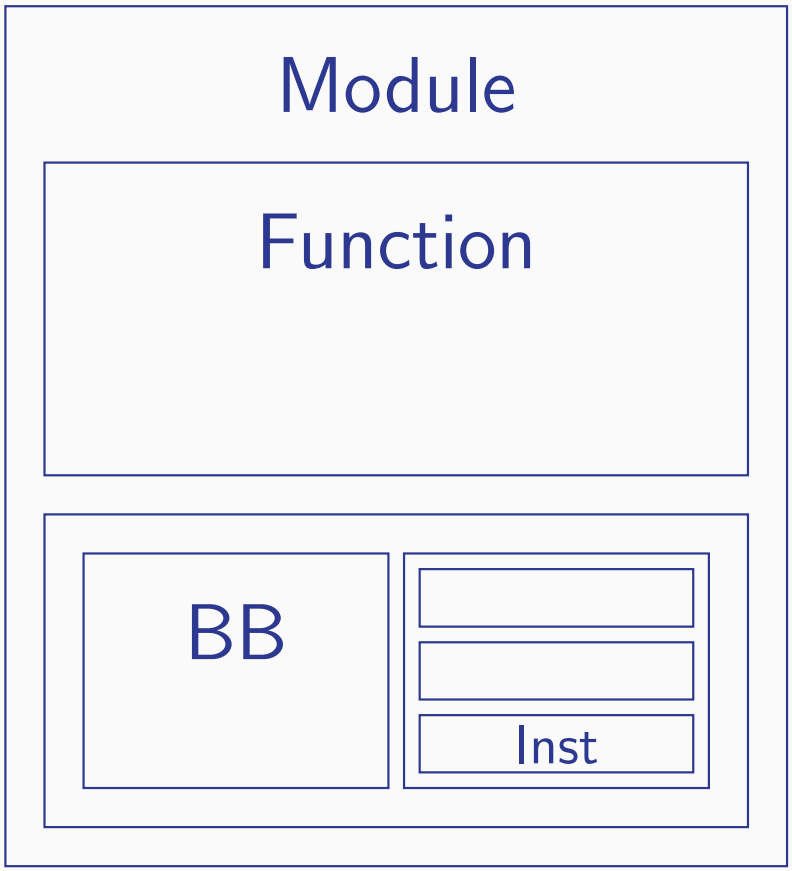
Instruction Booklet



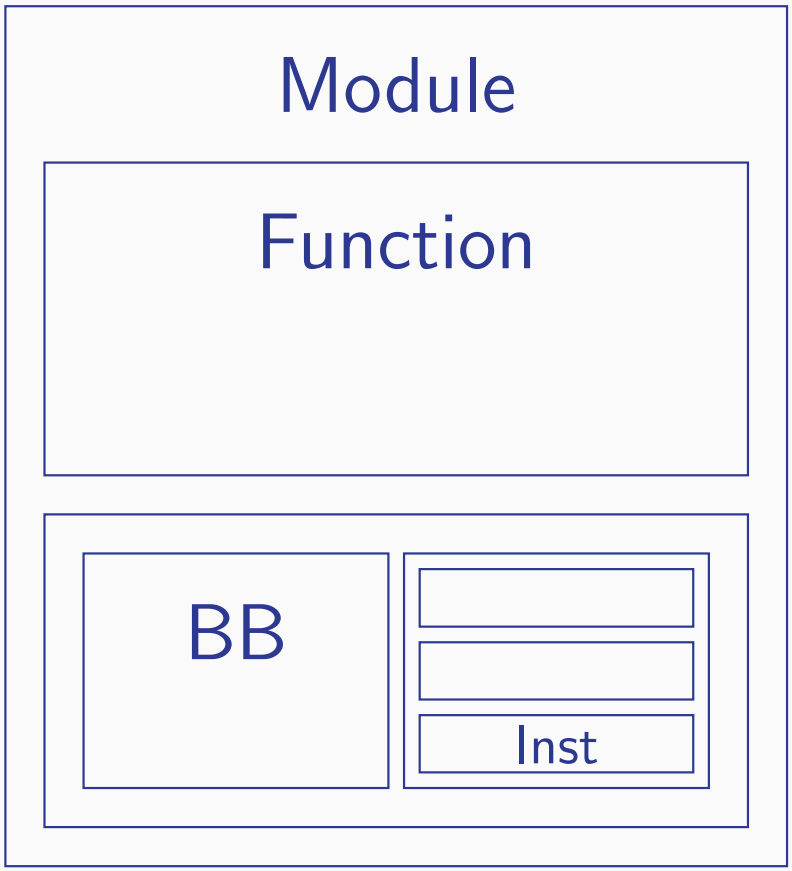
Instruction Booklet

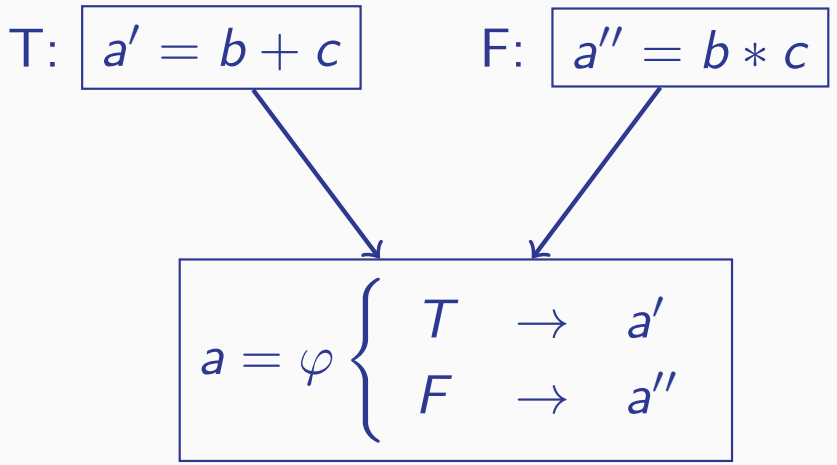


Instruction Booklet

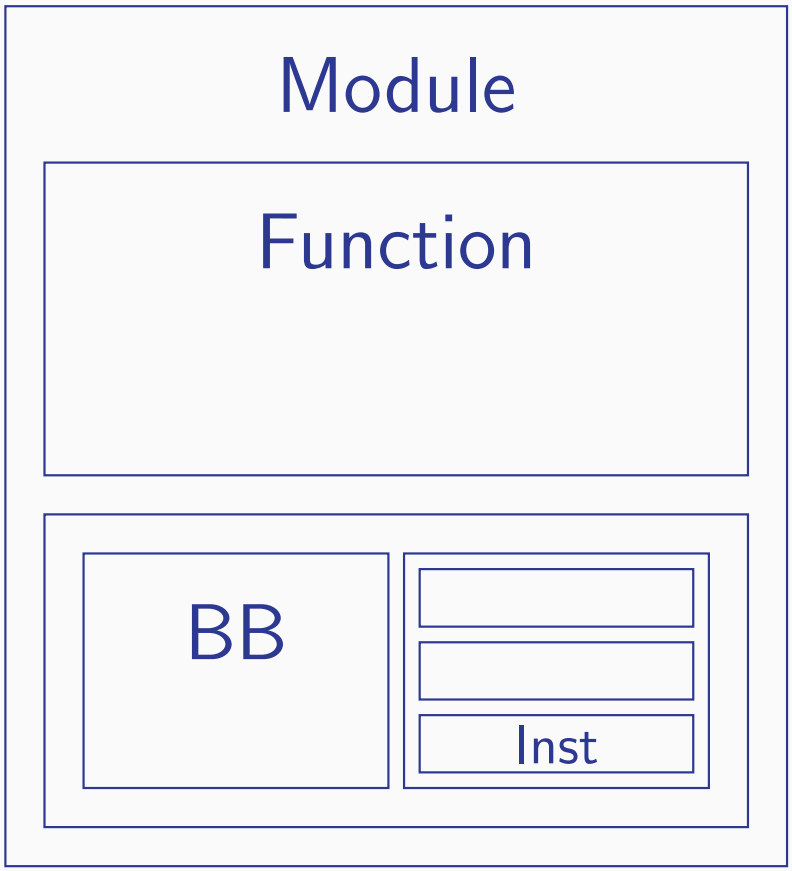


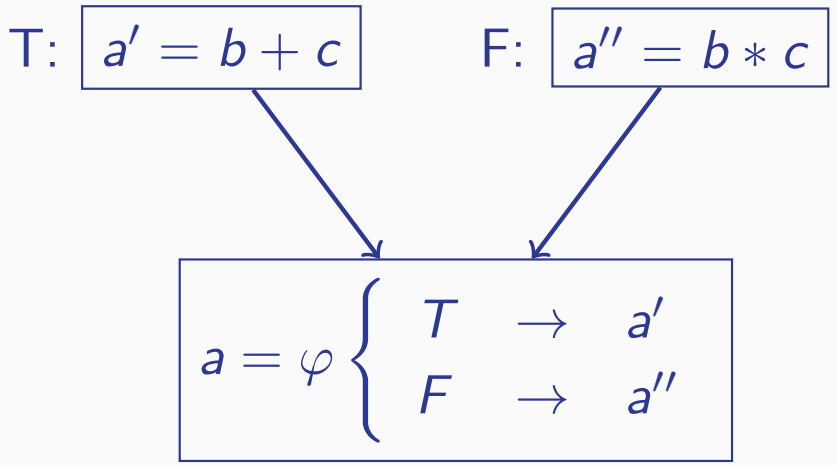
Instruction Booklet





Instruction Booklet





Press Start Button

LLVM 3.7 — Prerequisite

Please Load LLVM3.7

LLVM 3.7

Select difficulty

> **Easy** <

Hard

Nightmare

Stage Selection

Adding a new Front-End

In-Tree Pass Development

> **Out-of-Tree Pass Development** <

Adding a new Back-End

LLVM 3.7

OS Selection

> **Linux** <

OSX

Windows

Level Up

Stage 1 — Build Setup

Stage 2

Stage 3

Stage 4

stage 1

Setup a Proper CMake Project

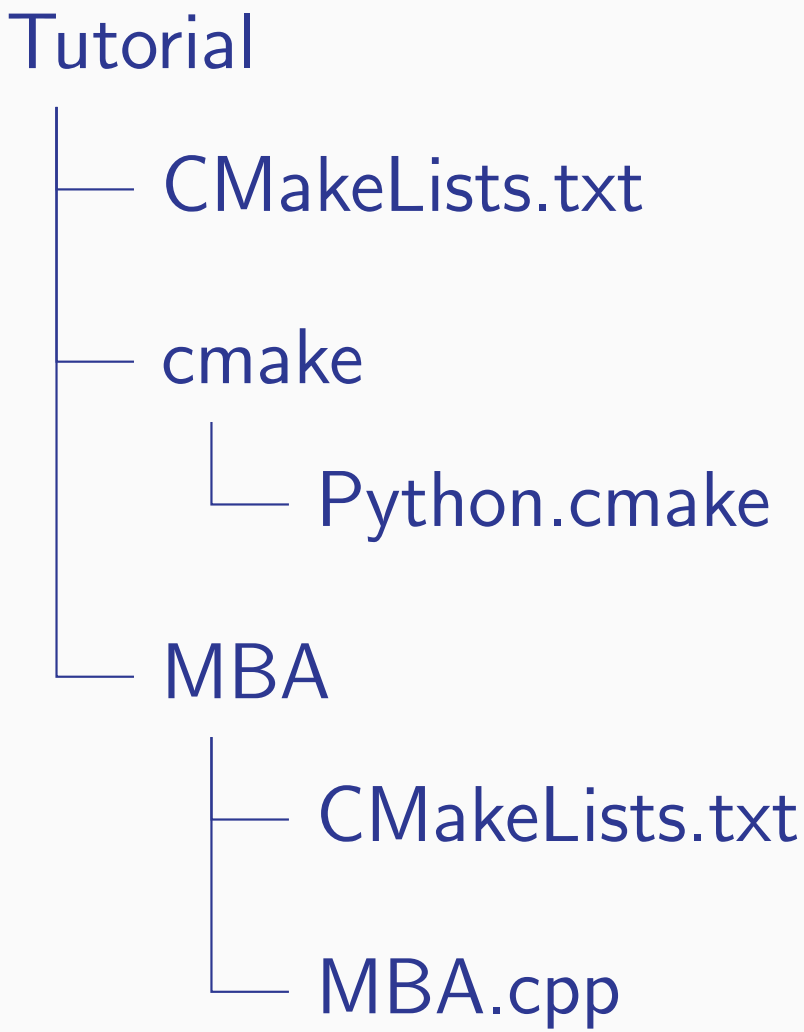
Goals

- Use LLVM CMake support
- Build a minimal pass

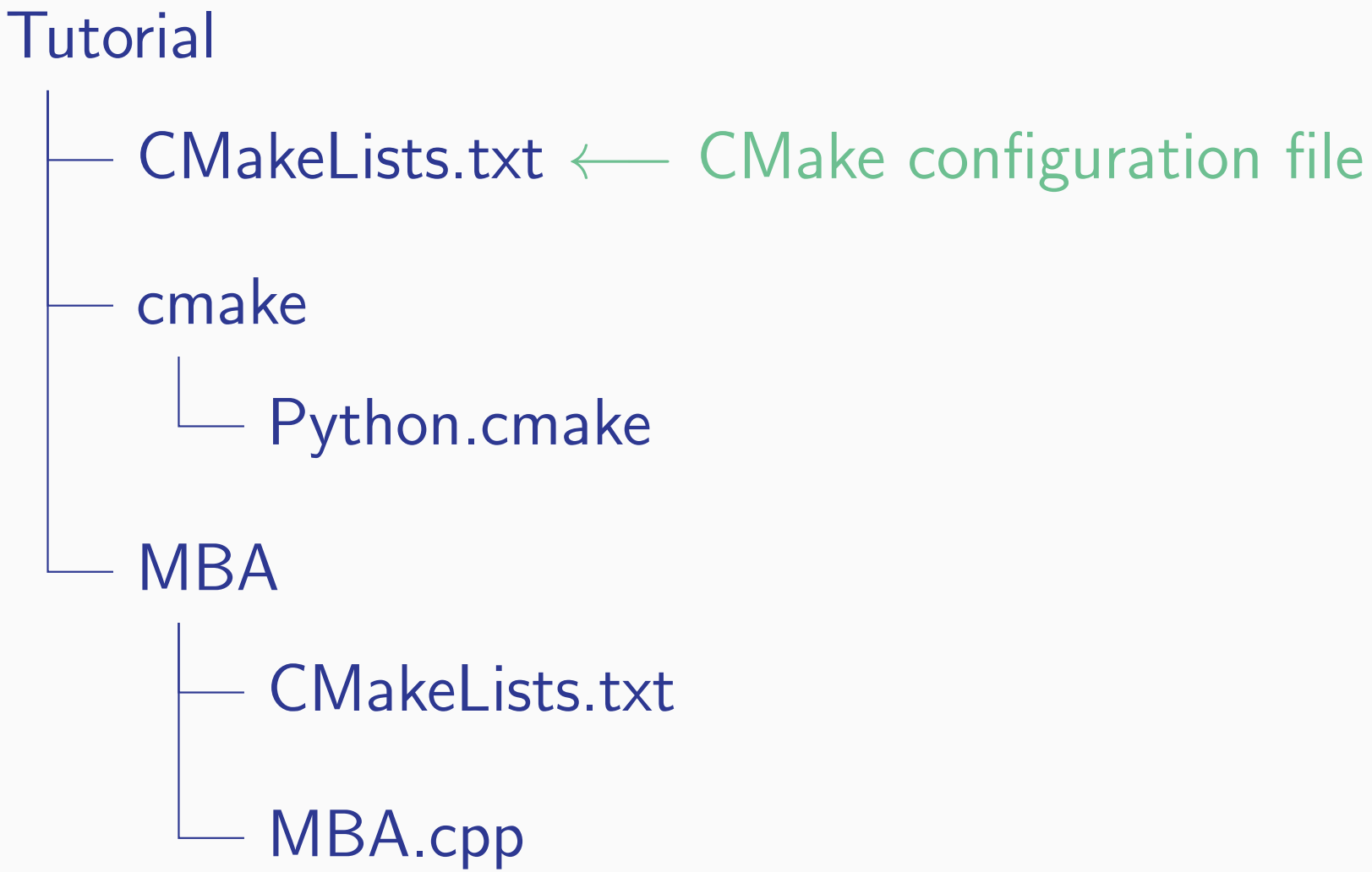
Bonus

- Setup a minimal test driver
- Make the pass compatible with `clang`

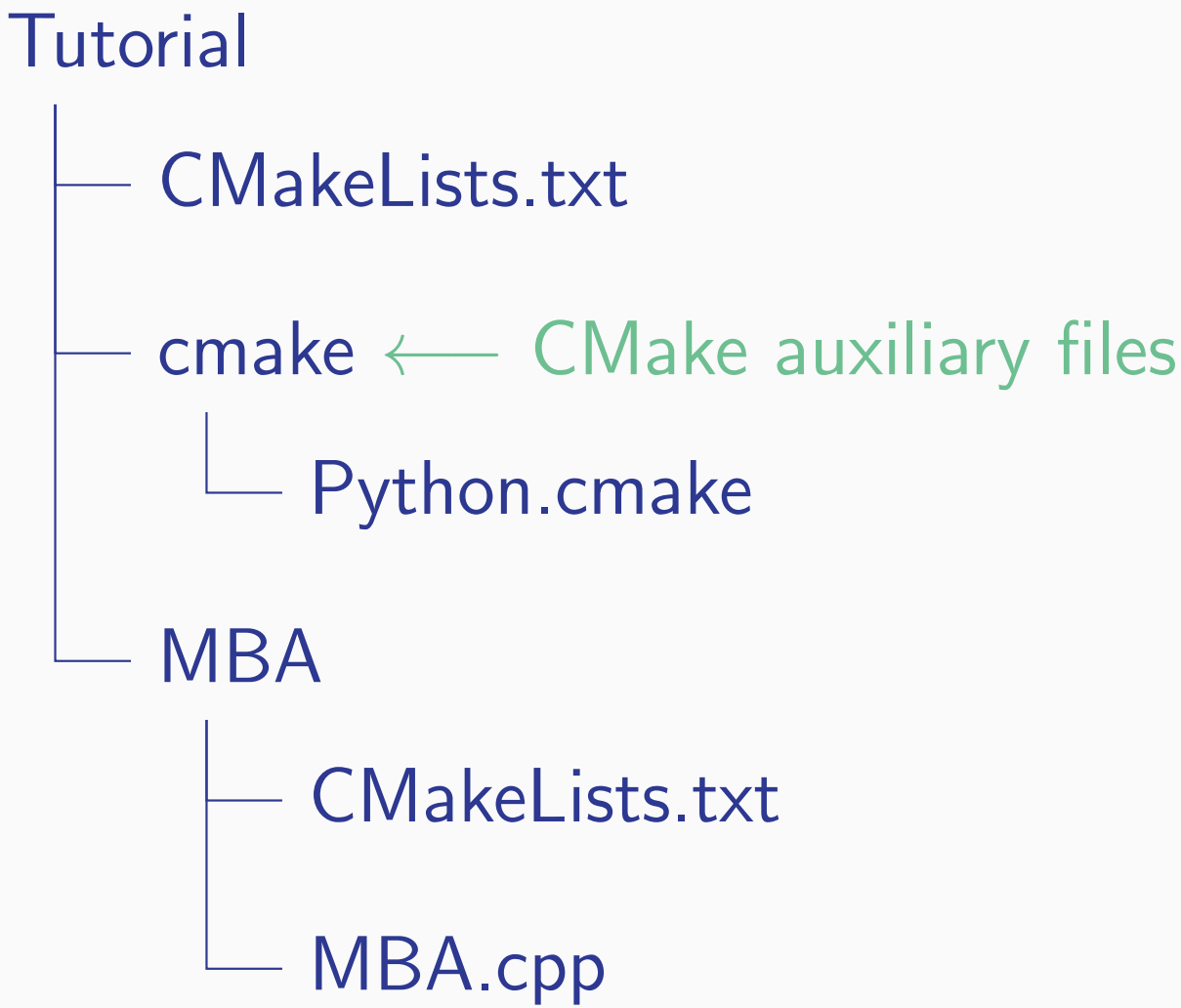
stage 1 — Directory Layout



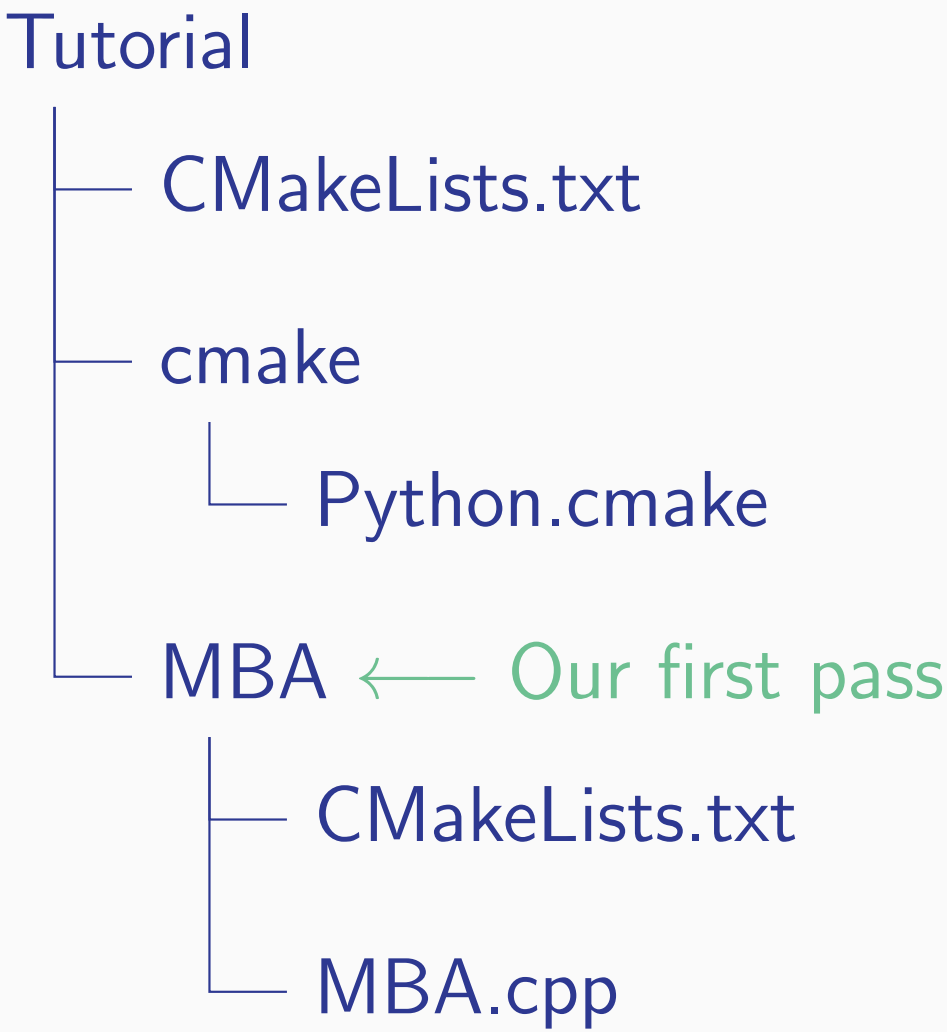
stage 1 — Directory Layout



stage 1 — Directory Layout



stage 1 — Directory Layout



stage 1 — CMakeLists.txt

LLVM Detection

```
set(LLVM_ROOT "" CACHE PATH "Root of LLVM install.")

# A bit of a sanity check:
if(NOT EXISTS ${LLVM_ROOT}/include/llvm )
    message(FATAL_ERROR
            "LLVM_ROOT (${LLVM_ROOT}) is invalid")
endif()
```

stage 1 — CMakeLists.txt

Load LLVM Config

```
list(APPEND CMAKE_PREFIX_PATH
     "${LLVM_ROOT}/share/llvm/cmake")
find_package(LLVM REQUIRED CONFIG)
```

And more LLVM Stuff

```
list(APPEND CMAKE_MODULE_PATH "${LLVM_CMAKE_DIR}")
include(HandleLLVMOptions) # load additional config
include(AddLLVM) # used to add our own modules
```

stage 1 — CMakeLists.txt

Propagate LLVM setup to our project

```
add_definitions(${LLVM_DEFINITIONS})
include_directories(${LLVM_INCLUDE_DIRS})
# See commit r197394, needed by add_llvm_module in llvm
    /CMakeLists.txt
set(LLVM_RUNTIME_OUTPUT_INTDIR "${CMAKE_BINARY_DIR}/bin
    /${CMAKE_CFG_INT_DIR}")
set(LLVM_LIBRARY_OUTPUT_INTDIR "${CMAKE_BINARY_DIR}/lib
    /${CMAKE_CFG_INT_DIR}")
```

Get Ready!

```
add_subdirectory(MBA)
```

stage 1 — MBA/CMakeLists.txt

Declare a Pass

```
add_llvm_loadable_module(LLVMMBA MBA.cpp)
```

1 Pass = 1 Dynamically Loaded Library

- Passes are loaded by a pass driver: opt

```
% opt -load LLVMMBA.so -mba foo.ll -S
```

- Or by clang (provided an extra setup)

```
% clang -Xclang -load -Xclang LLVMMBA.so foo.c -c
```

stage 1 — MBA.cpp

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
using namespace llvm;
  MBA() : BasicBlockPass(ID)
  {}
  bool runOnBasicBlock(BasicBlock &BB) override {
    bool modified = false;
    return modified;
  }
};
```

stage 1 — MBA.cpp

Registration Stuff

- Only performs registration for opt use!
- Uses a static constructor...

```
static RegisterPass<MBA>
    X("mba",  // the option name -> -mba
      "Mixed Boolean Arithmetic Substitution", //
         option description
      true, // true as we don't modify the CFG
      false // true if we're writing an analysis
      );
```

stage 1 — Bonus Level

Setup test infrastructure

- Rely on `lit`, LLVM's Integrated Tester
- `% pip install --user lit`

`CMakeLists.txt` update

```
list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_SOURCE_DIR}/cmake")
include(Python)
find_python_module(lit REQUIRED)
add_custom_target(check
    COMMAND ${PYTHON_EXECUTABLE} -m lit.main
            "${CMAKE_CURRENT_BINARY_DIR}/Tests" -v
    DEPENDS LLVMMBA LLVMReachableIntegerValues LLVMDuplicateBB
)
```

stage 1 — Bonus Level

Make the pass usable from `clang`

- Automatically loaded in `clang`'s optimization flow:

  `clang -Xclang -load -Xclang`

- Several extension points exist

```
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/Transforms/IPO/PassManagerBuilder.h"

static void registerClangPass(const PassManagerBuilder &,
                              legacy::PassManagerBase &PM)
{ PM.add(new MBA()); }
static RegisterStandardPasses RegisterClangPass
    (PassManagerBuilder::EP_EarlyAsPossible, registerClangPass);
```

Level Up

Stage 1

Stage 2 — Simple Pass

Stage 3

Stage 4

stage 2

Build a Simple Pass

Goals

- Learn basic LLVM IR manipulations
- Write a simple test case

Bonus

- Collect statistics on your pass
- Collect debug informations on your pass

stage 2 — MBA

Mixed Boolean Arithmetic

Simple Instruction Substitution

Turns: $a + b$

Into: $(a \oplus b) + 2 \times (a \wedge b)$

Context

$\Rightarrow$ Useful for code obfuscation

stage 2 — runOnBasicBlock++

- Iterate over a BasicBlock
- Use LLVM's dyn_cast to check the instruction kind

```
for (auto IIT = BB.begin(), IE = BB.end(); IIT !=
    IE; ++IIT) {
  Instruction &Inst = *IIT;
  auto *BinOp = dyn_cast<BinaryOperator>(&Inst);
  if (!BinOp)
    continue;
  unsigned Opcode = BinOp->getOpcode();
  if (Opcode != Instruction::Add || !BinOp->getType
      ()->isIntegerTy())
```

stage 2 — runOnBasicBlock++

LLVM Instruction creation/insertion:

- Use `IRBuilder` from `llvm/IR/IRBuilder.h`
- Creates $(a \oplus b) + 2 \times (a \wedge b)$

```
IRBuilder<> Builder(BinOp);
Value *NewValue = Builder.CreateAdd(
  Builder.CreateXor(BinOp->getOperand(0),
                    BinOp->getOperand(1)),
  Builder.CreateMul(
    ConstantInt::get(BinOp->getType(), 2),
    Builder.CreateAnd(
      BinOp->getOperand(0),
      BinOp->getOperand(1)))
);
```

stage 2 — `runOnBasicBlock++`

Instruction substitution:

- Use `llvm::ReplaceInstWithValue` that does the job for you (need to be careful on iterator validity)

```
ReplaceInstWithValue(BB.getInstList(),
                     IIT, NewValue);
```

stage 2 — Write a simple test

`lit` principles

- One source file (say `.c` or `.ll`) per test case
- Use comments to describe the test
- Use substitution for test configuration

`FileCheck` — `grep` on steroids!

- Compares `argv[1]` and `stdin`
- Reads check**s** from comments in `argv[1]`

$\Rightarrow$ Requires LLVM with `-DLLVM_INSTALL_UTILS`

stage 2 — Tests

```
// RUN: clang %s -O2 -S -emit-llvm -o %t.ll
// RUN: opt -load %bindir/lib/LLVMMBA${MOD_EXT} -mba %t
    .ll -S -o %t0.ll
// RUN: FileCheck %s < %t0.ll
// RUN: clang %t0.ll -o %t0
// RUN: %t0 -42 42
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[]) {
  if(argc != 3)
    return 1;
  int a = atoi(argv[1]),
      b = atoi(argv[2]);
// CHECK: and
  return a + b;
}
```

stage 2 — More tests

```
; RUN: opt -load %bindir/lib/LLVMMBA${MOD_EXT} -mba -mba-ratio=1 %s
    -S | FileCheck -check-prefix=CHECK-ON %s
; RUN: opt -load %bindir/lib/LLVMMBA${MOD_EXT} -mba -mba-ratio=0 %s
    -S | FileCheck -check-prefix=CHECK-OFF %s

; CHECK-LABEL: @foo(
define i32 @foo(i32 %i, i32 %j) {

...

; CHECK-ON: mul
; CHECK-OFF-NOT: mul
  %add = add i32 %i.addr.0, %j

...

}
```

stage 2 — Bonus

Collect Statistics

How many substitutions have we done?

```
#include "llvm/ADT/Statistic.h"
STATISTIC(MBACount, "The # of substituted instructions"
    );

...

 ++MBACount;
```

Collect them!

```
% opt -load LLVMMBA.so -mba -stats ...
```

stage 2 — Bonus

Debug your pass

DEBUG() and DEBUG_TYPE

Setup a guard:

```
#define DEBUG_TYPE "mba"
#include "llvm/Support/Debug.h"
```

Add a trace:

```
DEBUG(dbgs() << *BinOp << " -> " << *NewValue << "\n");
```

Collect the trace

```
% opt -O2 -mba -debug ... # verbose
% opt -O2 -mba -debug-only=mba ... # selective
```

Level Up

Stage 1

Stage 2

Stage 3 — Analyse

Stage 4

stage 3

Build an Analysis

Goals

- Use Dominator trees
- Write a `llvm::FunctionPass`
- Describe dependencies

Bonus

- Follow LLVM's guidelines

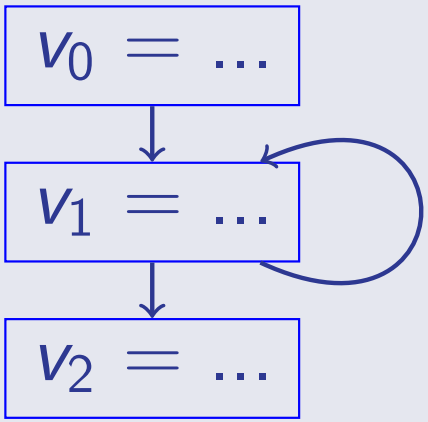
stage 3 — ReachableIntegerValues

Simple Module Analyse

Create a mapping between a `BasicBlock` and a set of `Values` that can be used in this block.

Algorithm

V = Visible values, D = Defined Values



$$V = \varnothing, D = \{v_0\}$$

$$V = \{v_0\}, D = \{v_1\}$$

$$V = \{v_0, v_1\}, D = \{v_2\}$$

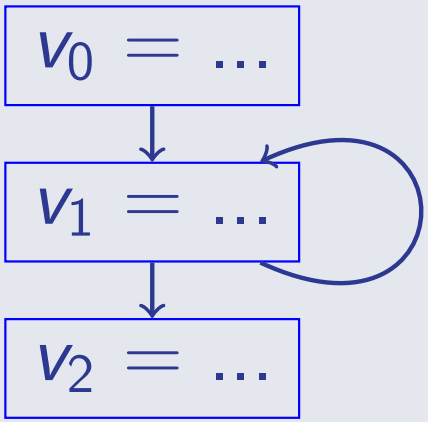
stage 3 — ReachableIntegerValues

Simple Module Analyse

Create a mapping between a `BasicBlock` and a set of `Values` that can be used in this block.

Algorithm

V = Visible values, D = Defined Values



$$V = \varnothing, D = \{v_0\}$$

$$V = \{v_0\}, D = \{v_1\}$$

$$V = \{v_0, v_1\}, D = \{v_2\}$$

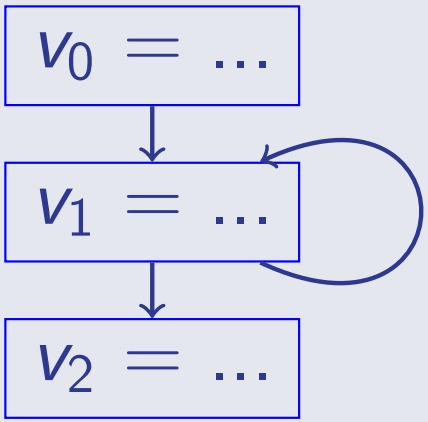
stage 3 — ReachableIntegerValues

Simple Module Analyse

Create a mapping between a `BasicBlock` and a set of `Values` that can be used in this block.

Algorithm

V = Visible values, D = Defined Values



$$V = \varnothing, D = \{v_0\}$$

$$V = \{v_0\}, D = \{v_1\}$$

$$V = \{v_0, v_1\}, D = \{v_2\}$$

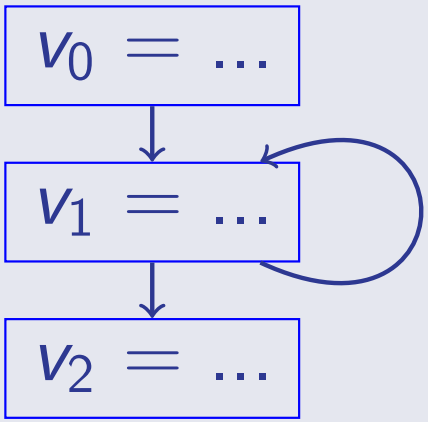
stage 3 — ReachableIntegerValues

Simple Module Analyse

Create a mapping between a `BasicBlock` and a set of `Values` that can be used in this block.

Algorithm

V = Visible values, D = Defined Values



$$V = \varnothing, D = \{v_0\}$$

$$V = \{v_0\}, D = \{v_1\}$$

$$V = \{v_0, v_1\}, D = \{v_2\}$$

stage 3 — Building an Analysis

Pass Registration

```
static RegisterPass<ReachableIntegerValuesPass>
    X("reachable-integer-values",          // pass option
      "Compute Reachable Integer values", // pass description
      true, // does not modify the CFG
      true  // and it's an analysis
      );
```

CMakeLists.txt

```
add_llvm_loadable_module(LLVMReachableIntegerValues
    ReachableIntegerValues.cpp)
```

stage 3 — Analysis

- Need to export the class declaration in a header
- Need to load the analysis in opt explicitly
- Result of the analysis stored as a member variable

API

```
void getAnalysisUsage(llvm::AnalysisUsage &Info)
    const override;
bool runOnFunction(llvm::Function &) override;
ReachableIntegerValuesMapTy const &
    getReachableIntegerValuesMap() const;
```

stage 3 — Make Result Available

Dependency Processing

1. PM runs each required analysis (if not cached)
2. PM runs the Pass entry point
3. The Pass calls `getAnalysis<...>` to access the instance

stage 3 — Declare Dependencies

Dependency on DominatorTree

```
void ReachableIntegerValuesPass::getAnalysisUsage(
    AnalysisUsage &Info) const {
  Info.addRequired<DominatorTreeWrapperPass>();
  Info.setPreservesAll();
}
```

stage 3 — runOnFunction

Entry Point

```
bool ReachableIntegerValuesPass::runOnFunction(Function
    &F) {
  ReachableIntegerValuesMap.clear();

    //...init stuff

  auto *Root =
      getAnalysis<DominatorTreeWrapperPass>().
          getDomTree().getRootNode();

    //...fill the map

  return false;
}
```

stage 3 — Bonus

LLVM's coding standard

Optional: You're working out-of tree.

But...

- Provides a common reference
- Helps for visual consistency

```
% find . \( -name '*.cpp' -o -name '*.h' \) \
    -exec clang-format-3.7 -i {} \;
```

http://llvm.org/docs/CodingStandards.html

Level Up

Stage 1

Stage 2

Stage 3

Stage 4 — Complex Pass

stage 4

Write a Complex Pass

Goals

- Use φ nodes
- Modify the Control Flow Graph (CFG)

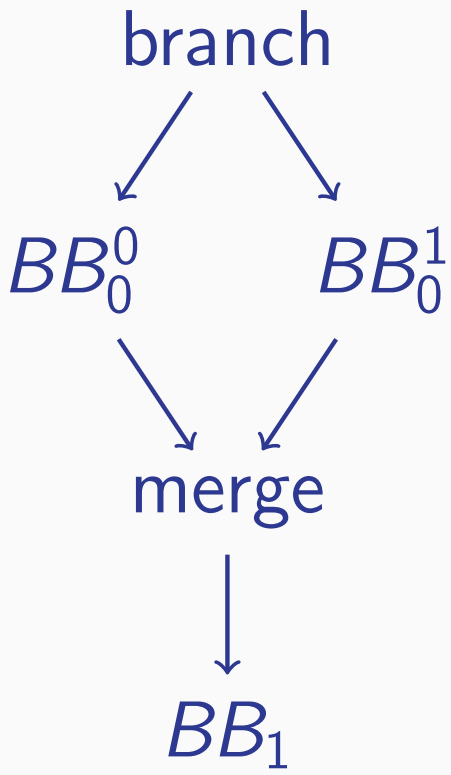
Bonus

- Declare extra options
- Fuzz your passes
- Add a support library

stage 4 — Duplicate Basic Blocks

Before

After



stage 4 — Problems

- Cloning `BasicBlocks` and iterating over a function loops
- Cloning an instruction creates a new `Value`
- Cloning several instructions requires a remapping

stage 4 — Forge a Random Branch

Get analysis result

```
auto const &RIV = getAnalysis<ReachableIntegerValuesPass>()
                      .getReachableIntegerValuesMap();
```

Pick a random reachable value

```
std::uniform_int_distribution<size_t> Dist(0, ReachableValuesCount-1)
auto Iter = ReachableValues.begin();
std::advance(Iter, Dist(RNG));
```

Random condition

```
Value *Cond = Builder.CreateIsNull(
    ReMapper.count(ContextValue) ?
    ReMapper[ContextValue] :
    ContextValue);
```

stage 4 — Messing with Clones

Cloning an instruction

```
Instruction *ThenClone = Instr.clone(),
            *ElseClone = Instr.clone();
```

Remap operands

```
RemapInstruction(ThenClone, ThenVMap, RF_IgnoreMissingEntries);
```

Manual φ creation

```
PHINode *Phi = PHINode::Create(ThenClone->getType(), 2);
Phi->addIncoming(ThenClone, ThenTerm->getParent());
Phi->addIncoming(ElseClone, ElseTerm->getParent());
```

stage 4 — Bonus

Fuzz your creation

Using `csmith`

1. Pick `http://embed.cs.utah.edu/csmith/`
2. Write a configuration file, e.g. `fuzz.cfg`:

```
clang -O2
clang -O2 -Xclang -load -Xclang LLVMDuplicateBB.so
```

3. Run generation!

```
% CSMITH_HOME=$PWD ./scripts/compiler_test.pl 1000 fuzz.cfg
```


stage 4 — Bonus

Add extra options

Control the obfuscation ratio

```
static llvm::cl::opt<Ratio> DuplicateBBRatio{
    "duplicate-bb-ratio",
    llvm::cl::desc("Only apply the duplicate basic block "
                   "pass on <ratio> of the basic blocks"),
    llvm::cl::value_desc("ratio"),
    llvm::cl::init(1.),
    llvm::cl::Optional
};
```

⇒ Need to specialize `llvm:cl` for the `Ratio` class.

stage 4 — Bonus

Add a support library

CMakeLists.txt

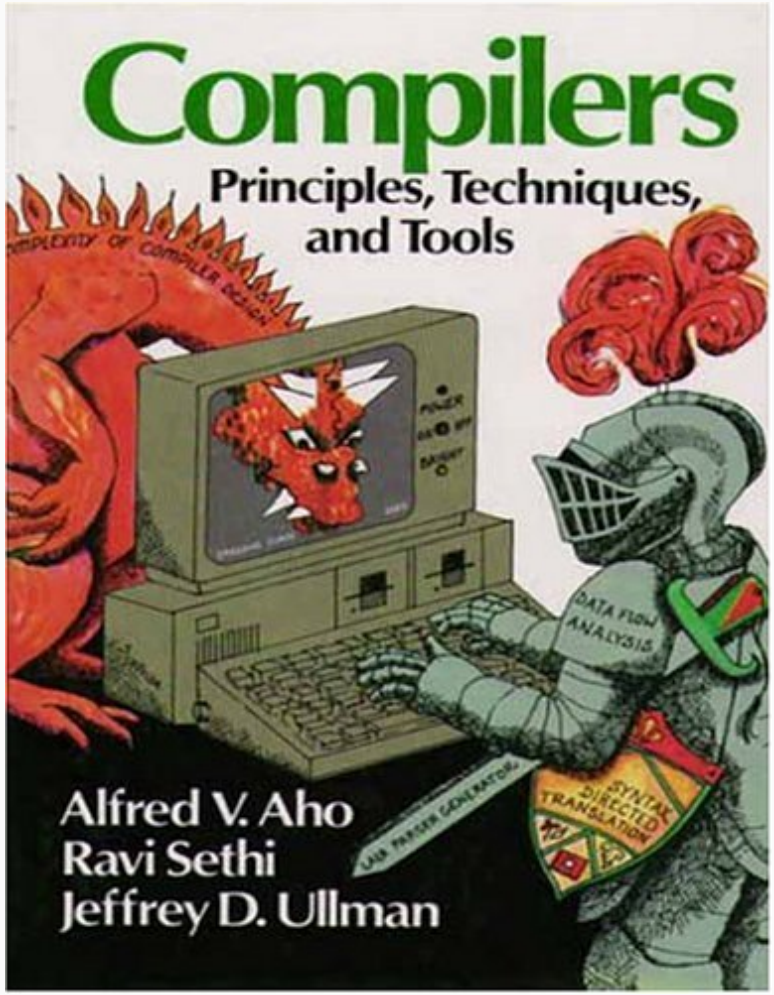
```
target_link_libraries(LLVMDuplicateBB Utils)
```

Specialize `llvm::cl::parser`

```
namespace llvm {
namespace cl {

template <> class parser<Ratio> : public basic_parser<Ratio> {
```

Final Boss


Compilers
Principles, Techniques, and Tools
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman
DATA FLOW ANALYSIS
SYNTAX DIRECTED TRANSLATION

Final Boss


DRAGON PUNCH
Compilers
Principles, Techniques, and Tools
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

GAME OVER

Creditz

Serge Guelton <sguelton@quarkslab.com>

Adrien Guinet <aguinet@quarkslab.com>

https://github.com/quarkslab/
llvm-dev-meeting-tutorial-2015

Insert Coins

Exit

> Play Again <