

LLVM & Clang

LLVM : Low Level Virtual Machine

Presented By Sumit Lahiri¹

¹IIT Kanpur

Hands On Session for LLVM & clang

- LLVM : Low Level Virtual Machine.

- LLVM : Low Level Virtual Machine.
- Compiler Infrastructure.

- LLVM : Low Level Virtual Machine.
- Compiler Infrastructure.
- Frontend : clang, C++, C, go, java to AST and finally to LLVM-IR.

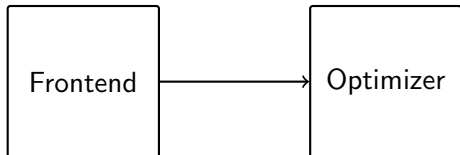
- LLVM : Low Level Virtual Machine.
- Compiler Infrastructure.
- Frontend : clang, C++, C, go, java to AST and finally to LLVM-IR.
- Middle End : opt tool, Optimizations and other passes on LLVM-IR.

- LLVM : Low Level Virtual Machine.
- Compiler Infrastructure.
- Frontend : clang, C++, C, go, java to AST and finally to LLVM-IR.
- Middle End : opt tool, Optimizations and other passes on LLVM-IR.
- Back End : LLVM CodeGen/Backend, LLVM-IR to target code generator.

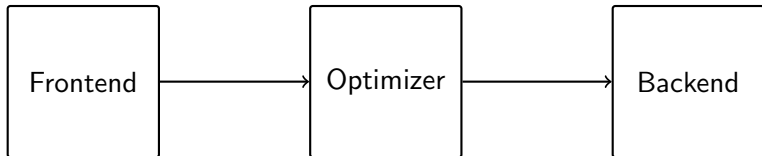
Compilers



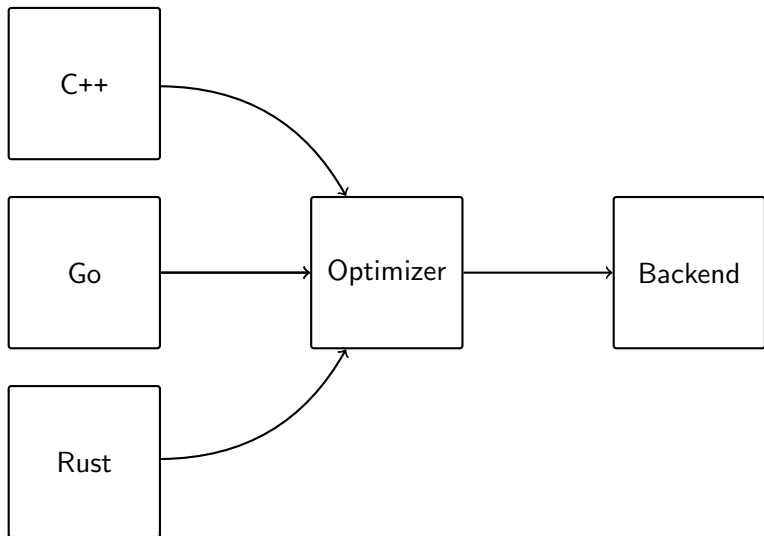
Frontend



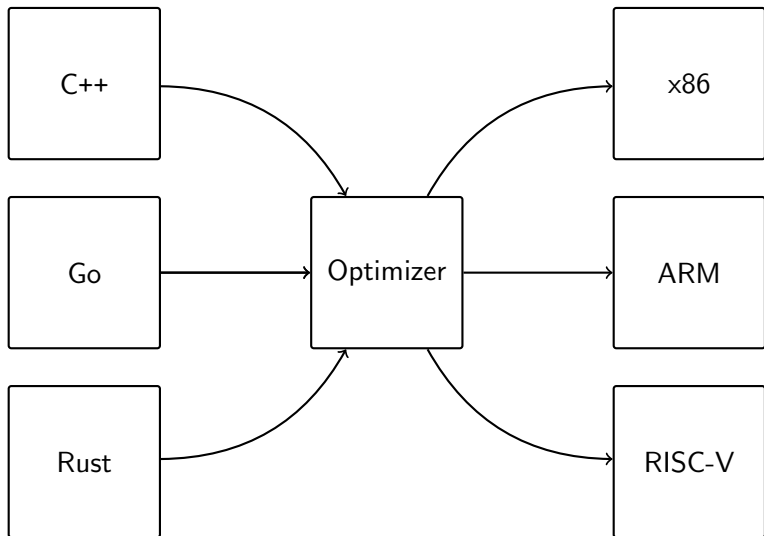
Compilers

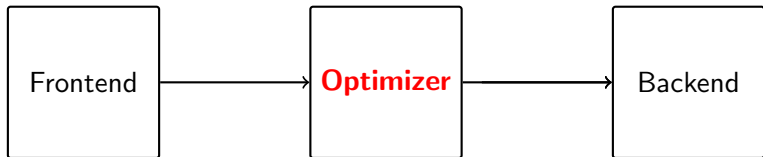


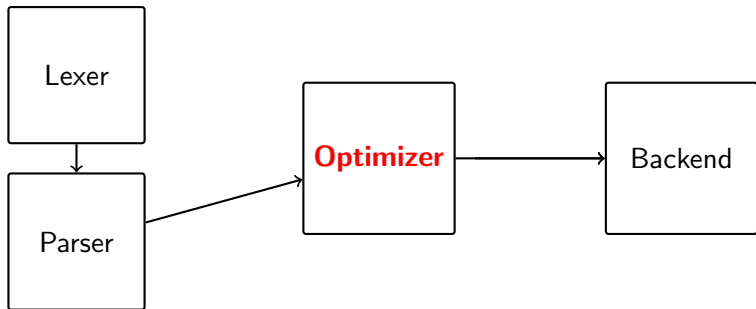
Compilers



Compilers

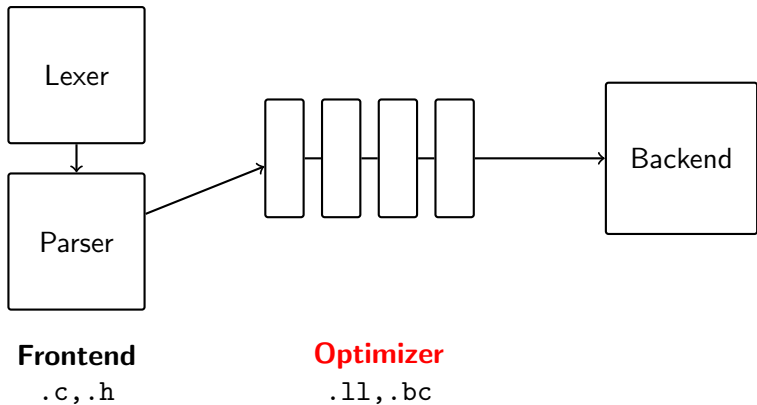


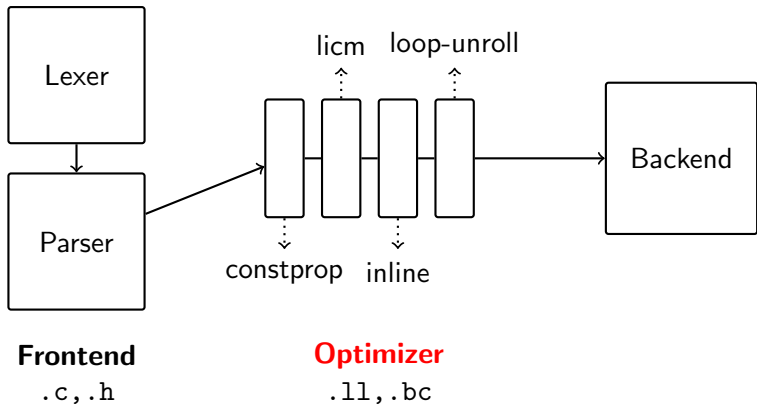


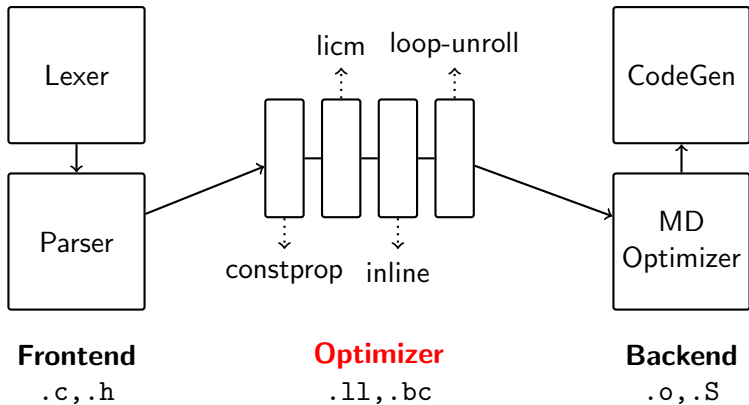


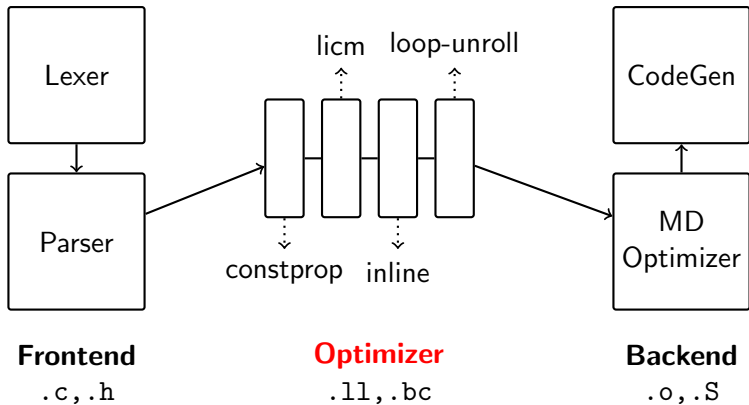
Frontend

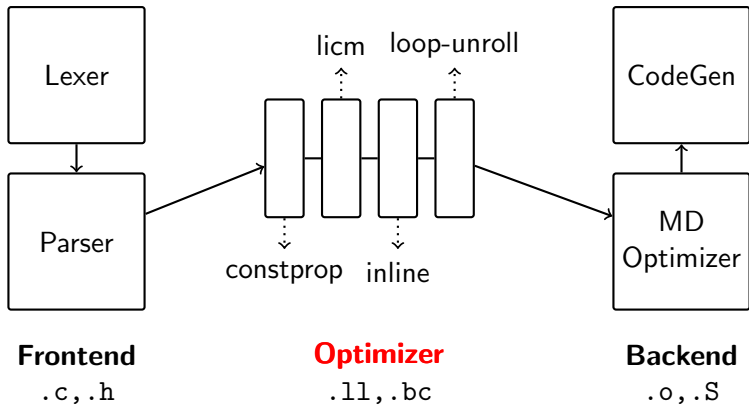
`.c, .h`











Clang AST

```
1  int retsum(int a, int b) {  
2      return a + b;  
3  }  
4  # clang -Xclang -ast-dump -fsyntax-only code.cpp  
5  [-FunctionDecl <test.cc:1:1, line:3:1> retsum 'int (int, int)'  
6      |-ParmVarDecl <col:12, col:16> col:16 used a 'int'  
7      |-ParmVarDecl <col:19, col:23> col:23 used b 'int'  
8          [-CompoundStmt <col:26, line:3:1>  
9              [-ReturnStmt <line:2:3, col:14>  
10                  [-BinaryOperator <col:10, col:14> 'int' '+'  
11                      |-ImplicitCastExpr <col:10> 'int' <LValueToRValue>  
12                          | [-DeclRefExpr <col:10> 'int' lvalue ParmVar 'a' 'int'  
13                              [-ImplicitCastExpr <col:14> 'int' <LValueToRValue>  
14                                  [-DeclRefExpr <col:14> 'int' lvalue ParmVar 'b' 'int'
```

Clang AST

```
1  int main()
2  {
3      int a = 90;
4  }
5  # ./clang_ast "int main() { int a = 90; }"
6  FunctionDecl 0x3705db0 <input.cc:1:1, col:26> col:5 main ['int ()']
7      [-CompoundStmt 0x3705f78 <col:12, col:26>
8          [-DeclStmt 0x3705f60 <col:14, col:24>
9              [-VarDecl 0x3705ed8 <col:14, col:22> col:18 a ['int'] cinit
10                 [-IntegerLiteral 0x3705f40 <col:22> ['int'] 90
```

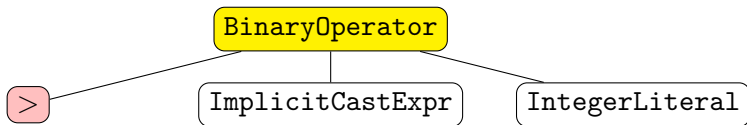
```
x > 0xff2
```

```
| -BinaryOperator 0x3c32fb0 '_Bool' '>'  
| | -ImplicitCastExpr 0x3c32f98 'int' <LValueToRValue>  
| | | -DeclRefExpr 0x3c32f58 'int' lvalue Var 0x3c32ed8 'x' 'int'  
| | -IntegerLiteral 0x3c32f78 'int' 4082
```

BinaryOperator

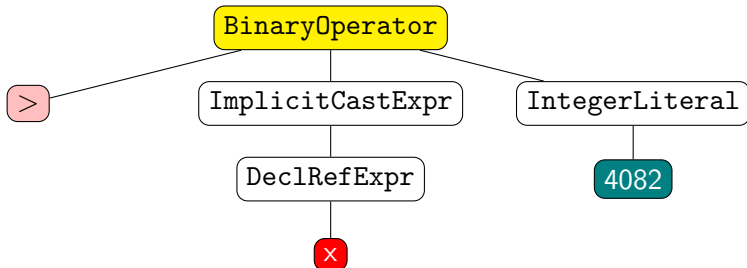
```
x > 0xff2
```

```
| -BinaryOperator 0x3c32fb0 '_Bool' '>'  
| | -ImplicitCastExpr 0x3c32f98 'int' <LValueToRValue>  
| | | -DeclRefExpr 0x3c32f58 'int' lvalue Var 0x3c32ed8 'x' 'int'  
| | -IntegerLiteral 0x3c32f78 'int' 4082
```



```
x > 0xff2
```

```
| -BinaryOperator 0x3c32fb0 '_Bool' '>'  
| | -ImplicitCastExpr 0x3c32f98 'int' <LValueToRValue>  
| | | -DeclRefExpr 0x3c32f58 'int' lvalue Var 0x3c32ed8 'x' 'int'  
| | -IntegerLiteral 0x3c32f78 'int' 4082
```



LLVM IR

```
1  int retsum(int a, int b) {
2      return a + b;
3  }
4  # clang -S -emit-llvm code.cpp -O0 -o code.ll
5  ; ModuleID = 'test.cc'
6  source_filename = "test.cc"
7  target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:..."
8  target triple = "x86_64-unknown-linux-gnu"
9
10 ; Function Attrs: mustprogress noline nounwind optnone uwtable
11 define dso_local i32 @Z6retsumii(i32 %0, i32 %1) #0 {
12     %3 = alloca i32, align 4
13     %4 = alloca i32, align 4
14     store i32 %0, i32* %3, align 4
15     store i32 %1, i32* %4, align 4
16     %5 = load i32, i32* %3, align 4
17     %6 = load i32, i32* %4, align 4
18     %7 = add nsw i32 %5, %6
19     ret i32 %7
20 }
```

LLVM Pass

- Primarily analyze and transform the LLVM intermediate representation.

LLVM Pass

- Primarily analyze and transform the LLVM intermediate representation.
- Eg : Run your own optimization on the LLVM IR.

LLVM Pass

- Primarily analyze and transform the LLVM intermediate representation.
- Eg : Run your own optimization on the LLVM IR.
- Get the IR representation using `-S -emit-llvm` flags.

LLVM Pass

- Primarily analyze and transform the LLVM intermediate representation.
- Eg : Run your own optimization on the LLVM IR.
- Get the IR representation using `-S -emit-llvm` flags.
- Transformation Pass – Modify the CFG. Extract Useful info.
`-licm`, `-dce`

LLVM Pass

- Primarily analyze and transform the LLVM intermediate representation.
- Eg : Run your own optimization on the LLVM IR.
- Get the IR representation using `-S -emit-llvm` flags.
- Transformation Pass – Modify the CFG. Extract Useful info.
`-licm`, `-dce`
- Analysis Pass – Run analysis on the BBs of CFG. `-domtree`,
`-dot-cfg`

LLVM Pass

- Primarily analyze and transform the LLVM intermediate representation.
- Eg : Run your own optimization on the LLVM IR.
- Get the IR representation using `-S -emit-llvm` flags.
- Transformation Pass – Modify the CFG. Extract Useful info.
`-licm`, `-dce`
- Analysis Pass – Run analysis on the BBs of CFG. `-domtree`, `-dot-cfg`
- Utility Pass – View/Log some information from CFG.

LLVM Pass

- Primarily analyze and transform the LLVM intermediate representation.
- Eg : Run your own optimization on the LLVM IR.
- Get the IR representation using `-S -emit-llvm` flags.
- Transformation Pass – Modify the CFG. Extract Useful info.
`-licm`, `-dce`
- Analysis Pass – Run analysis on the BBs of CFG. `-domtree`, `-dot-cfg`
- Utility Pass – View/Log some information from CFG.
- Run on Module

LLVM Pass

- Primarily analyze and transform the LLVM intermediate representation.
- Eg : Run your own optimization on the LLVM IR.
- Get the IR representation using `-S -emit-llvm` flags.
- Transformation Pass – Modify the CFG. Extract Useful info.
`-licm`, `-dce`
- Analysis Pass – Run analysis on the BBs of CFG. `-domtree`, `-dot-cfg`
- Utility Pass – View/Log some information from CFG.
- Run on Module or Function.

LLVM Pass

- Primarily analyze and transform the LLVM intermediate representation.
- Eg : Run your own optimization on the LLVM IR.
- Get the IR representation using `-S -emit-llvm` flags.
- Transformation Pass – Modify the CFG. Extract Useful info.
`-licm`, `-dce`
- Analysis Pass – Run analysis on the BBs of CFG. `-domtree`, `-dot-cfg`
- Utility Pass – View/Log some information from CFG.
- Run on Module or Function.
 - Code up pass logic in struct inherited from `PassInfoMixin`,

LLVM Pass

- Primarily analyze and transform the LLVM intermediate representation.
- Eg : Run your own optimization on the LLVM IR.
- Get the IR representation using `-S -emit-llvm` flags.
- Transformation Pass – Modify the CFG. Extract Useful info.
`-licm`, `-dce`
- Analysis Pass – Run analysis on the BBs of CFG. `-domtree`, `-dot-cfg`
- Utility Pass – View/Log some information from CFG.
- Run on Module or Function.
 - Code up pass logic in struct inherited from `PassInfoMixin`, must have a `run()` function.

LLVM Pass

- Primarily analyze and transform the LLVM intermediate representation.
- Eg : Run your own optimization on the LLVM IR.
- Get the IR representation using `-S -emit-llvm` flags.
- Transformation Pass – Modify the CFG. Extract Useful info.
`-licm`, `-dce`
- Analysis Pass – Run analysis on the BBs of CFG. `-domtree`, `-dot-cfg`
- Utility Pass – View/Log some information from CFG.
- Run on Module or Function.
 - Code up pass logic in struct inherited from `PassInfoMixin`, must have a `run()` function.
 - Register the Pass and build your pass into a shared library which can be loaded and used by `opt` tool to run pass on LLVM IR.

LLVM Pass

```
1  struct MyPass : public PassInfoMixin<MyPass> {
2      PreservedAnalyses run(Function &F, FunctionAnalysisManager &FM){
3          # Your code logic
4          ...
5          return PreservedAnalyses::all();
6      }
7      ...
8  };
9  extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_WEAK
10 llvmGetPassPluginInfo() {
11     return {LLVM_PLUGIN_API_VERSION, "MyPass", "v0.1",
12             [] (PassBuilder &PB) {
13                 PB.registerPipelineParsingCallback(
14                     [] (StringRef Name, FunctionPassManager &FPM,
15                         ArrayRef<PassBuilder::PipelineElement>) {
16                         if (Name == "mypass") {
17                             FPM.addPass(ModifyBuildCFG());
18                             return true;
19                         }
20                         return false; }); }); });
21 }
```

LLVM Pass

```
1  struct MyPass : public PassInfoMixin<MyPass> {
2      PreservedAnalyses run(Module &T, ModuleAnalysisManager &M){
3          # Your code logic
4          ...
5          return PreservedAnalyses::all();
6      }
7      ...
8  };
9  extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_WEAK
10 llvmGetPassPluginInfo() {
11     return {LLVM_PLUGIN_API_VERSION, "MyPass", "v0.1",
12             [] (PassBuilder &PB) {
13                 PB.registerPipelineParsingCallback(
14                     [] (StringRef Name, ModulePassManager &MPM,
15                         ArrayRef<PassBuilder::PipelineElement>) {
16                         if (Name == "mypass") {
17                             MPM.addPass(ModifyBuildCFG());
18                             return true;
19                         }
20                         return false; }); });
21 }
```

Clang Frontend Pass AKA ClangAST Frontend Action

- Input is the source code, i.e. C/C++/go file.

Clang Frontend Pass AKA ClangAST Frontend Action

- Input is the source code, i.e. C/C++/go file.
- Abstract Syntax Tree which can be traversed.

Clang Frontend Pass AKA ClangAST Frontend Action

- Input is the source code, i.e. C/C++/go file.
- Abstract Syntax Tree which can be traversed.
- Run a FrontEnd action on the AST.

Clang Frontend Pass AKA ClangAST Frontend Action

- Input is the source code, i.e. C/C++/go file.
- Abstract Syntax Tree which can be traversed.
- Run a FrontEnd action on the AST.
- Representation : Stmt,

Clang Frontend Pass AKA ClangAST Frontend Action

- Input is the source code, i.e. C/C++/go file.
- Abstract Syntax Tree which can be traversed.
- Run a FrontEnd action on the AST.
- Representation : Stmt, Decl

Clang Frontend Pass AKA ClangAST Frontend Action

- Input is the source code, i.e. C/C++/go file.
- Abstract Syntax Tree which can be traversed.
- Run a FrontEnd action on the AST.
- Representation : Stmt, Decl or Expr.

Clang Frontend Pass AKA ClangAST Frontend Action

- Input is the source code, i.e. C/C++/go file.
- Abstract Syntax Tree which can be traversed.
- Run a FrontEnd action on the AST.
- Representation : Stmt, Decl or Expr.
- Start from the TopLevelDecl or TranslationUnitDecl

Clang Frontend Pass AKA ClangAST Frontend Action

- Input is the source code, i.e. C/C++/go file.
- Abstract Syntax Tree which can be traversed.
- Run a FrontEnd action on the AST.
- Representation : Stmt, Decl or Expr.
- Start from the TopLevelDecl or TranslationUnitDecl and recursively parse down.

Clang Frontend Pass AKA ClangAST Frontend Action

- Input is the source code, i.e. C/C++/go file.
- Abstract Syntax Tree which can be traversed.
- Run a FrontEnd action on the AST.
- Representation : Stmt, Decl or Expr.
- Start from the TopLevelDecl or TranslationUnitDecl and recursively parse down.
- Clang Plugin or Standalone tool (clang LibTooling).

Clang Frontend Pass AKA ClangAST Frontend Action

- `ASTFrontendAction` : Interface to define Action to be performed on the AST.

Clang Frontend Pass AKA ClangAST Frontend Action

- `ASTFrontendAction` : Interface to define Action to be performed on the AST.
- `ASTConsumer` : Consumes the AST, `ASTFrontendAction` creates a consumer.

Clang Frontend Pass AKA ClangAST Frontend Action

- `ASTFrontendAction` : Interface to define Action to be performed on the AST.
- `ASTConsumer` : Consumes the AST, `ASTFrontendAction` creates a consumer.
- Handles what function to run or what to do with each `TranslationUnit`.

Clang Frontend Pass AKA ClangAST Frontend Action

- `ASTFrontendAction` : Interface to define Action to be performed on the AST.
- `ASTConsumer` : Consumes the AST, `ASTFrontendAction` creates a consumer.
- Handles what function to run or what to do with each `TranslationUnit`.
- `RecursiveASTVisitor` : Consumer can use a visitor to visit each `Decl Node` and perform certain actions.

Clang Frontend Pass AKA ClangAST Frontend Action

- `ASTFrontendAction` : Interface to define Action to be performed on the AST.
- `ASTConsumer` : Consumes the AST, `ASTFrontendAction` creates a consumer.
- Handles what function to run or what to do with each `TranslationUnit`.
- `RecursiveASTVisitor` : Consumer can use a visitor to visit each `Decl Node` and perform certain actions.
- Finally we build our logic into a tool using `CommonOptionsParser` & `ClangTool`.

Clang Frontend Pass AKA ClangAST Frontend Action

- `ASTFrontendAction` : Interface to define Action to be performed on the AST.
- `ASTConsumer` : Consumes the AST, `ASTFrontendAction` creates a consumer.
- Handles what function to run or what to do with each `TranslationUnit`.
- `RecursiveASTVisitor` : Consumer can use a visitor to visit each `Decl Node` and perform certain actions.
- Finally we build our logic into a tool using `CommonOptionsParser` & `ClangTool`.

Clang ASTFrontendAction

```
1  class ClassAction : public clang::ASTFrontendAction {
2      public:
3          # returns a uniq ptr to your consumer.
4          virtual std::unique_ptr<clang::ASTConsumer>
5              CreateASTConsumer(clang::CompilerInstance &Compiler,
6                               llvm::StringRef InFile) {
7              return
8                  # Instantiate your consumer.
9                  std::make_unique<ClassConsumer>(
10                      &Compiler.getASTContext()
11                  );
12      }
13  };
```

Clang ClassConsumer

```
1  class ClassConsumer : public clang::ASTConsumer {
2      public:
3          explicit ClassConsumer(ASTContext *Context)
4              : Visitor(Context) {}
5          virtual void HandleTranslationUnit(clang::ASTContext &Context) {
6              # Called on each TranslationDeclUnit
7              Visitor.TraverseDecl(Context.getTranslationUnitDecl());
8          }
9      private:
10         # Implements the actual recursive visit strategy.
11         ClassVisitor Visitor;
12     };
```

Clang ClassConsumer

```
1  class ClassVisitor
2  : public RecursiveASTVisitor<ClassVisitor> {
3  public:
4      explicit FindNamedClassVisitor(ASTContext *Context)
5          : Context(Context) {}
6
7      bool VisitWhileStmt(WhileStmt *S) {
8          llvm::outs() << "While Condition : ";
9          if (S)
10             VisitDecl(S->getConditionVariable());
11         return true;
12     }
13     # ... More Visit Logic.
14     bool VisitDecl(clang::Decl *Declaration) {
15         Declaration->dump();
16         return true;
17     }
18
19 private:
20     ASTContext *Context;
21 };
```


Questions??

Questions??

- Docker Image :
<https://hub.docker.com/r/prodrelworks/llvm-examples>
- LLVM Example :
<https://github.com/lahiri-phdworks/LLVM-Examples>
- Slide Diagram (LLVM/Clang) :
<https://github.com/peter-can-talk/cppnow-2017peter-can-talk>, see his repo, it's covers some cool topics!